

## FORMATION PAGE

Form Approved  
OPM No. 0704-0188Public  
release  
for read  
the ON

AD-A226 896

hour per response, including the time for reviewing instructions, searching existing data sources gathering and  
rewards regarding this burden estimate or any other aspect of this collection of information, including suggestions  
formation Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to  
Washington, DC 20503.

1. AGE

DATE

2. REPORT TYPE AND DATES COVERED

Final 09 May 1990 to 09 May 1991

4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report: SYSTEAM KG  
SYSTEAM Ada Compiler VAX/VMS 1.82, VAX 8530 (Host & Target),  
900509I1.11009

5. FUNDING NUMBERS

6. AUTHOR(S)

IABG-AVF

Ottobrunn, FEDERAL REPUBLIC OF GERMANY

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

IABG-AVF, Industrieanlagen-Betriebsgesellschaft  
Dept. SZT  
Einsteinstrasse 20  
D-8012 Ottobrunn  
FEDERAL REPUBLIC OF GERMANY8. PERFORMING ORGANIZATION  
REPORT NUMBER

IABG-VSR-065

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office  
United States Department of Defense  
Washington, D.C. 20301-308110. SPONSORING/MONITORING AGENCY  
REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

SYSTEAM KG, SYSTEAM Ada Compiler VAX/VMS 1.82, Ottobrunn, West Germany, VAX 8530 under  
VMS, Version 5.1 (Host & Target), ACVC 1.11.DTIC  
ELECTE  
SEP 25 1990  
S D14. SUBJECT TERMS Ada programming language, Ada Compiler Validation  
Summary Report, Ada Compiler Validation Capability, Validation  
Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-  
STD-1815A, Ada Joint Program Office

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION  
OF REPORT  
UNCLASSIFIED18. SECURITY CLASSIFICATION  
OF THIS PAGE  
UNCLASSIFIED19. SECURITY CLASSIFICATION  
OF ABSTRACT  
UNCLASSIFIED

20. LIMITATION OF ABSTRACT

## CHAPTER 1

### INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation-dependent but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

(KR) ←

AVF Control Number: IABG-VSR-065  
11 June 1990

Ada COMPILER  
VALIDATION SUMMARY REPORT:  
Certificate Number: #900509I1.11009  
SYSTEM KG  
SYSTEM Ada Compiler VAX/VMS 1.82  
VAX 8530 Host and Target

Prepared By:  
IABG mbH, Abt. ITE  
Einsteinstrasse 20  
D-8012 Ottobrunn  
West Germany

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 09 May 1990.

Compiler Name and Version: SYSTEAM Ada Compiler VAX/VMS  
Version 1.82

Host Computer System: VAX 8530 under VMS, Version 5.1

Target Computer System: VAX 8530 under VMS, Version 5.1

A more detailed description of this Ada implementation is found in section 3.1 of this report.

As a result of this validation effort, Validation Certificate ~~199050911.11000~~ is awarded to SYSTEAM KG. This certificate expires on 01 June 1992.

This report has been reviewed and is approved.

*Michael Tonndorf*

IABG mbH, Abt. ITE  
Michael Tonndorf  
Einsteinstrasse 20  
D-8012 Ottobrunn  
West Germany

Ada Validation Organization  
Director, Computer & Software  
Engineering Division  
Institute for Defense Analyses  
Alexandria VA 22311

*for Edward M. Lieberhardt*

Ada Joint Program Office  
Dr. John Solomond  
Director  
Department of Defense  
Washington DC 20301

Accession For	
NTS	ORAS
DTIC	TAB
Unpublished	
Justification	
By	
Distribution	
Availability Codes	
Dist	Availability Codes
A-1	

## Declaration of Conformance

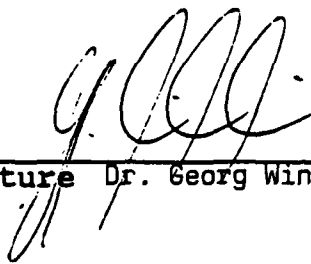
**Customer:** SYSTEAM KG  
**Ada Validation Facility:** IABG m. b. H., Abt. SZT  
**ACVC Version:** 1.11

### Ada Implementation

**Ada Compiler Name:** SYSTEAM Ada Compiler VAX/VMS  
**Version:** 1.82  
**Host Computer System:** VAX 8530 under VMS, Version 5.1  
**Target Computer System:** VAX 8530 under VMS, Version 5.1

### Customer's Declaration

I, the undersigned, representing SYSTEAM KG, declare that SYSTEAM KG has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation(s) listed in this declaration.

  
\_\_\_\_\_  
Signature Dr. Georg Winterstein

20.03.1990  
\_\_\_\_\_  
Date

## CONTENTS

CHAPTER	1	TEST INFORMATION . . . . .	1
	1.1	USE OF THIS VALIDATION SUMMARY REPORT . . . . .	1
	1.2	REFERENCES . . . . .	2
	1.3	ACVC TEST CLASSES . . . . .	2
	1.4	DEFINITION OF TERMS . . . . .	3
CHAPTER	2	IMPLEMENTATION DEPENDENCIES . . . . .	5
	2.1	WITHDRAWN TESTS . . . . .	5
	2.2	INAPPLICABLE TESTS . . . . .	5
	2.3	TEST MODIFICATIONS . . . . .	8
CHAPTER	3	PROCESSING INFORMATION . . . . .	9
	3.1	TESTING ENVIRONMENT . . . . .	9
	3.2	TEST EXECUTION . . . . .	10
APPENDIX	A	MACRO PARAMETERS	
APPENDIX	B	COMPILATION SYSTEM OPTIONS	
APPENDIX	C	APPENDIX F OF THE Ada STANDARD	

## CHAPTER 1

## INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro89] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro89]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

## 1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service  
5285 Port Royal Road  
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization  
Institute for Defense Analyses  
1801 North Beauregard Street  
Alexandria VA 22311

## 1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
- [Pro89] Ada Compiler Validation Procedures, Version 2.0, Ada Joint Program Office, May 1989.
- [UG89] Ada Compiler Validation Capability User's Guide, 24 October 1989.

## 1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPRT13, and the procedure CHECK\_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK\_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK\_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly



some inapplicable tests (see Section 3.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

#### 1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	Functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.

Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro89].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

## CHAPTER 2

## IMPLEMENTATION DEPENDENCIES

## 2.1 WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 60 tests had been withdrawn by the Ada Validation Organization (AVO) at the time of validation testing. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 31 January 1990.

E28005C	B41308B	C45114A	C45612B	C45651A	C46022A
B49008A	A74006A	B83022B	B83022H	B83025B	B83025D
B83026B	C83026A	C83041A	C97116A	BA2011A	CB7001A
CB7001B	CB7004A	CC1223A	BC1226A	CC1226B	BC3009B
CD2A21E	CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A
CD2B15C	BD3006A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	BD8002A
BD8004C	CD9005A	CD9005B	CDA201E	CE2107I	CE2119B
CE3111C	CE3118A	CE3411B	CE3412B	CE3812A	CE3902B

Immediately before on-site testing the AVF was informed by the AVO that C98003B will be withdrawn, too. Therefore, this test was processed but its result was ignored.

## 2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. The inapplicability criteria for some tests are explained in documents issued by ISO and the AJ&O known as Ada Issues and commonly referenced in the format AI-dddd. For this implementation, the following tests were inapplicable for the reasons indicated; references to Ada Issues are included as appropriate.

C24113W..Y (3 tests) contain lines of length greater than 255 characters which are not supported by this implementation.

C34007P and C34007S are expected to raise CONSTRAINT\_ERROR. This

The following 21 tests check for the predefined type LONG\_INTEGER:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

C41401A is expected to raise CONSTRAINT\_ERROR for the evaluation of certain attributes, however, this implementation derives the values from the subtypes of the prefix at compile time as allowed by 11.6 (7) LRM. Therefore, elaboration of the prefix is not involved and CONSTRAINT\_ERROR is not raised.

C45346A declares an array of length integer'last/2 +1. This implementation raises the proper exception when the array is declared.

C45624A checks that the proper exception is raised if machine\_overflows is false for floating point types with digits 5.

C45624B checks that the proper exception is raised if machine\_overflows is false for floating point types with digits 6.

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a SYSTEM.MAX\_MANTISSA of 47 or greater.

C86001F recompiles package SYSTEM, making package TEXT\_IO, and hence package REPORT, obsolete. For this implementation, the package TEXT\_IO is dependent upon package SYSTEM.

B86001Y checks for a predefined fixed-point type other than DURATION.

C96005B checks for values of type DURATION'BASE that are outside the range of DURATION. There are no such values for this implementation.

CD1009C uses a representation clause specifying a non-default size for a floating-point type (see AI-00561).

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

EE2401D contains instantiations of package DIRECT\_IO with unconstrained array types. This implementation raises USE\_ERROR upon creation of such a file.

The 21 tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO

CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	-----	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

CE2107B, CE2107E, CE2110B, and CE2111D attempt to associate multiple internal files with the same external file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

CE2107C, CE2107D, CE2107L, and CE2108B attempt to associate names with temporary sequential files. The proper exception is raised when such an association is attempted.

CE2107G, CE2110D, and CE2111H attempt to associate multiple internal files with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

CE2107H and CE2108D attempt to associate names with temporary direct files. The proper exception is raised when such an association is attempted.

CE2203A checks that WRITE raises USE\_ERROR if the capacity of the external file is exceeded for SEQUENTIAL\_IO. This implementation does not restrict file capacity.

CE2403A checks that WRITE raises USE\_ERROR if the capacity of the external file is exceeded for DIRECT\_IO. This implementation does not restrict file capacity.

CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A attempt to associate multiple internal files with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

CE3112B attempts to associate names with temporary text files. The proper exception is raised when such an association is attempted.

CE3304A checks that USE\_ERROR is raised if a call to SET\_LINE\_LENGTH or SET\_PAGE\_LENGTH specifies a value that is inappropriate for external files. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT\_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

### 2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 17 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B24009A	B29001A	B38003A	B38009A	B38009B
B91001H	BC2001D	BC2001E	BC3204B	BC3205B	BC3205D

For the following tests a pragma ELABORATE for the package REPORT was added.

C83030C	C86007A
---------	---------

The following tests compile without error, as allowed by AI-00256 -- the units are illegal only with respect to units that they do not depend on. However, all errors are detected at link time. The AVO ruled that this is acceptable behavior.

BC3204C	BC3204D	BC3205C	BC3205D
---------	---------	---------	---------

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information and sales information about this Ada implementation system, see:

System KG Dr. Winterstein  
Am Rüppurrer Schloß 7  
D-7500 Karlsruhe 51  
West Germany

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

### 3.2 TEST EXECUTION

Version 1.11 of the ACVC comprises 4140 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 97 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation. Tests were compiled, linked and executed (as appropriate) using a single computer.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options.

Tests were compiled using the command

```
SAS compile/LOG 'file name'
```

and linked using the command

```
SAS link/LOG/EXE='test name' .
```

The command qualifier /LOG enforces the compiler and the linker to write additional messages onto the specified file.

Chapter B tests were compiled with the full listing option, /LIST. For several tests, complete listings were added using the option /LIST = 'test name'.LL.

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

During preparation of the VSR the AVF detected that the values of the macro parameters \$FLOAT\_NAME and \$NAME did not correspond to the set of predefined types as stated in the APPENDIX F, package STANDARD, supplied by the customer. Therefore seven tests were rerun. The tests demonstrated acceptable behavior.



## APPENDIX A

## MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The following macro parameters are defined in terms of the value V of \$MAX\_IN\_LEN which is the maximum input line length permitted for the tested implementation. For these parameters, Ada string expressions are given rather than the macro values themselves.

Macro Parameter	Macro Value
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	'" & (1..V/2 => 'A') & '"'
\$BIG_STRING2	'" & (1..V-1-V/2 => 'A') & '1' & '"'
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
\$MAX_STRING_LITERAL	'" & (1..V-2 => 'A') & '"'

## MACRO PARAMETERS

The following table contains the values for the remaining macro parameters.

Macro Parameter	Macro Value
\$MAX_IN_LEN	255
\$ACC_SIZE	32
\$ALIGNMENT	4
\$COUNT_LAST	2147483647
\$DEFAULT_MEM_SIZE	2147483648
\$DEFAULT_STOR_UNIT	8
\$DEFAULT_SYS_NAME	VAX_VMS
\$DELTA_DOC	2#1.0#E-31
\$ENTRY_ADDRESS	SYSTEM.INTERRUPT_VECTOR(10)
\$ENTRY_ADDRESS1	SYSTEM.INTERRUPT_VECTOR(11)
\$ENTRY_ADDRESS2	SYSTEM.INTERRUPT_VECTOR(12)
\$FIELD_LAST	512
\$FILE_TERMINATOR	' '
\$FIXED_NAME	NO_SUCH_FIXED_NAME
\$FLOAT_NAME	LONG_LONG_FLOAT
\$FORM_STRING	" "
\$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
\$GREATER_THAN_DURATION	0.0
\$GREATER_THAN_DURATION_BASE_LAST	200_000.0
\$GREATER_THAN_FLOAT_BASE_LAST	16#0.8#E+32
\$GREATER_THAN_FLOAT_SAFE_LARGE	16#0.7FFF_FFFF_1000_000#E+32
\$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	16#0.7FFF_FD0#E+32

## MACRO PARAMETERS

\$HIGH\_PRIORITY 15

\$ILLEGAL\_EXTERNAL\_FILE\_NAME1  
[NODIRECTORY]FILENAME

\$ILLEGAL\_EXTERNAL\_FILE\_NAME2  
FILENAME.\*

\$INAPPROPRIATE\_LINE\_LENGTH  
-1

\$INAPPROPRIATE\_PAGE\_LENGTH  
-1

\$INCLUDE\_PRAGMA1 PRAGMA INCLUDE ("A28006D1.TST")

\$INCLUDE\_PRAGMA2 PRAGMA INCLUDE ("B28006F1.TST")

\$INTEGER\_FIRST -2147483648

\$INTEGER\_LAST 2147483647

\$INTEGER\_LAST\_PLUS\_1 2147483648

\$INTERFACE\_LANGUAGE VMS

\$LESS\_THAN\_DURATION -0.0

\$LESS\_THAN\_DURATION\_BASE\_FIRST  
-200\_000.0

\$LINE\_TERMINATOR ' '

\$LOW\_PRIORITY 0

\$MACHINE\_CODE\_STATEMENT  
NULL;

\$MACHINE\_CODE\_TYPE NO\_SUCH\_TYPE

\$MANTISSA\_DOC 31

\$MAX\_DIGITS 33

\$MAX\_INT 2147483647

\$MAX\_INT\_PLUS\_1 2147483648

\$MIN\_INT -2147483648

\$NAME SHORT\_SHORT\_INTEGER

\$NAME\_LIST VAX\_VMS

# MACRO PARAMETERS

\$NAME_SPECIFICATION1	BIGB:[V182.ACVC11.CHAPE]X2120A.;1
\$NAME_SPECIFICATION2	BIGB:[V182.ACVC11.CHAPE]X2120B.;1
\$NAME_SPECIFICATION3	BIGB:[V182.ACVC11.CHAPE]X3119A.;1
\$NEG_BASED_INT	16#FFFFFFFE#
\$NEW_MEM_SIZE	2147483648
\$NEW_STOR_UNIT	8
\$NEW_SYS_NAME	VAX_VMS
\$PAGE_TERMINATOR	' '
\$RECORD_DEFINITION	NEW INTEGER
\$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
\$TASK_SIZE	32
\$TASK_STORAGE_SIZE	10240
\$TICK	0.01
\$VARIABLE_ADDRESS	GET_VARIABLE_ADDRESS
\$VARIABLE_ADDRESS1	GET_VARIABLE_ADDRESS1
\$VARIABLE_ADDRESS2	GET_VARIABLE_ADDRESS2
\$YOUR_PRAGMA	RESIDENT

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

## 4 Compiling

After a program library has been created, one or more compilation units can be compiled in the context of this library. The compilation units can be placed on different source files or they can all be on the same file. One unit, a parameterless procedure, acts as the main program. If all units needed by the main program and the main program itself have been compiled successfully, they can be linked. The resulting code can then be executed by giving a RUN command.

§4.1 and Chapter 5 describe in detail how to call the Compiler and the Linker. In §4.2 the Completer, which is called to generate code for instances of generic units, is described.

Chapter 6 explains the information which is given if the execution of a program is abandoned due to an unhandled exception.

The information the Compiler produces and outputs in the Compiler listing is explained in §4.4.

Finally, the log of a sample session is given in Chapter 7.

### 4.1 Compiling Ada Units

To start the SYSTEAM Ada Compiler, use the SAS COMPILE command.

SAS COMPILE	Command Description
<b>Format</b>	
\$ SAS COMPILE file-spec[,...]	
<b>Command Qualifiers</b>	<b>Defaults</b>
/[NO]ANALYZE_DEPENDENCY	/NOANALYZE_DEPENDENCY
/LIBRARY=directory-spec	/LIBRARY=[.ADALIB]
/[NO]LOG[=file-spec]	/NOLOG
/[NO]RECOMPILE	/NORECOMPILE
<b>Positional Qualifiers</b>	<b>Defaults</b>
/[NO]CHECK	/CHECK
/[NO]COPY_SOURCE	/COPY_SOURCE
/[NO]INLINE	/INLINE
/[NO]LIST[=file-spec]	/NOLIST
/[NO]MACHINE_CODE	/NOMACHINE_CODE
/[NO]OPTIMIZE	/OPTIMIZE

## Command Parameters

### **file-spec**

Specifies the file(s) to be compiled. The default directory is []. The default file type is ADA. The maximum length of lines in file-spec is 255. The maximum number of source lines in file-spec is 65534. Wild cards are allowed.

Note: If you specify a wild card the order of the compilation is alphabetical, which is not always successful. Thus wild cards should be used together with /ANALYZE\_DEPENDENCY. With this qualifier the sources can be processed in any order.

## Description

The source file may contain a sequence of compilation units (cf. LRM(§10.1)). All compilation units in the source file are compiled individually. When a compilation unit is compiled successfully, the program library is updated and the Compiler continues with the compilation of the next unit on the source file. If the compilation unit contained errors, they are reported (see §4.4). In this case, no update operation is performed on the program library and all subsequent compilation units in the compilation are only analyzed without generating code.

The Compiler delivers the status code WARNING on termination (cf. VAX/VMS, *DCL Dictionary*, command EXIT) if one of the compilation units contained errors. A message corresponding to this code has not been defined; hence %NONAME-W-NOMSG is printed upon notification of a batch job terminated with this status.

## Command Qualifiers

### **/ANALYZE\_DEPENDENCY**

### **/NOANALYZE\_DEPENDENCY (D)**

Specifies that the Compiler only performs syntactical analysis and the analysis of the dependencies on other units. The units in file-spec are entered into the library if they are syntactically correct. The actual compilation is done later with the SAS AUTOCOMPILE command.

Note: An already existing unit with the same name as the new one is replaced and all dependent units become obsolete, unless the source file of both are identical. In this case the library is *not* updated because the dependencies are already known.

By default, the normal, full compilation is done.

**/LIBRARY=dir-spec**

Specifies the program library the command works on. The SAS COMPILE command needs write access to the library. The default is [.ADALIB].

**/LOG[=file-spec]**

**/NOLOG (D)**

Controls whether the Compiler writes additional messages onto the specified file. The default file name is SYS\$OUTPUT. The default file type is LOG.

By default, no additional messages are written.

**/RECOMPILE**

**/NORECOMPILE (D)**

Indicates that a recompilation of a previously analyzed source is to be performed. This qualifier should not be used unless the command was produced by the SYSTEAM Ada Recompiler. See the SAS RECOMPILE command.

## Positional Qualifiers

**/CHECK (D)**

**/NOCHECK**

Controls whether all run-time checks are suppressed. If you specify /NOCHECK this is equivalent to the use of PRAGMA suppress for all kinds of checks.

By default, no run-time checks are suppressed, except in cases where PRAGMA suppress\_all appears in the source.

**/COPY\_SOURCE (D)**

**/NOCOPY\_SOURCE**

Controls whether a copy of the source file is kept in the library. The copy in the program library is used for later access by the Debugger or tools like the Recompiler. The name of the copy is generated by the Compiler and need normally not be known by the user. The Recompiler and the Debugger know this name. You can use the SAS DIRECTORY/FULL command to see the file name of the copy. If a specified file contains several compilation units a copy containing only the source text of one compilation unit is stored in the library for each compilation unit. Thus the Recompiler can recompile a single unit.

If /NOCOPY\_SOURCE is specified, the Compiler only stores the name of the source file in the program library. In this case the Recompiler and the Debugger are able to use the original file if it still exists.



`/COPY_SOURCE` cannot be specified together with `/ANALYZE_DEPENDENCY`.

`/INLINE (D)`

`/NOINLINE`

Controls whether inline expansion is performed as requested by `PRAGMA inline`. If you specify `/NOINLINE` these pragmas are ignored.

By default, inline expansion is performed.

`/LIST[=file-spec]`

`/NOLIST (D)`

Controls whether a listing file is created. One listing file is created for each source file compiled. If `/LIST` is placed as a command qualifier a listing file is created for all sources. If `/LIST` is placed as a parameter qualifier a listing file is created only for the corresponding source file.

The default directory for listing files is the current default directory. The default file name is the name of the source file being compiled unless `/RE-compile` is specified. In this case the name of the original source file, which is stored in the library, is taken as default. The default file type is `LIS`. No wildcard characters are allowed in the file specification.

By default, the `COMPILE` command does not create a listing file.

`/MACHINE_CODE`

`/NOMACHINE_CODE (D)`

Controls whether machine code is appended at the listing file. `/MACHINE_CODE` has no effect if `/NOLIST` or `/ANALYZE_DEPENDENCY` is specified.

By default, no machine code is appended at the listing file.

`/OPTIMIZE (D)`

`/NOOPTIMIZE`

Controls whether full optimization is applied in generating code. There is no way to specify that only certain optimizations are to be performed.

By default, full optimization is done.

---

End of Command Description

---

## 4.2 Completing Generic Instances

Since the Compiler does not generate code for instances of generic bodies, the Completer must be used to complete such units before a program using the instances can be executed. The Completer must also be used to complete packages in the program which do not require a body. This is done implicitly when the Linker is called.

It is also possible to call the Completer explicitly with the SAS COMPLETE command.

---

### SAS COMPLETE

---

### Command Description

---

#### Format

```
$ SAS COMPLETE unit[,...]
```

#### Command Qualifiers

/[NO]CHECK	/CHECK
/[NO]INLINE	/INLINE
/LIBRARY=directory-spec	/LIBRARY=[ ADALIB]
/[NO]LIST[=file-spec]	/NOLIST
/[NO]LOG[=file-spec]	/NOLOG
/[NO]MACHINE_CODE	/NOMACHINE_CODE
/[NO]OPTIMIZE	/OPTIMIZE

#### Defaults

#### Command Parameters

**unit**

specifies the unit(s) whose execution closure is to be completed.

#### Description

The SAS COMPLETE command invokes the SYSTEAM Ada Completer. The Completer generates code for all instantiations of generic units in the execution closure of the specified unit(s). It also generates code for packages without bodies (if necessary).

By default, the Completer is invoked implicitly by the SAS LINK command. In normal cases there is no need to invoke it explicitly.

#### Command Qualifiers

/CHECK (D)

/NOCHECK

Controls whether all run-time checks are suppressed. If you specify /NOCHECK this is equivalent to the use of PRAGMA suppress for all kinds of checks.

By default, no run-time checks are suppressed, except in cases where PRAGMA suppress\_all appears in the source.

**/INLINE (D)**

**/NOINLINE**

Controls whether inline expansion is performed as requested by PRAGMA inline. If /NOINLINE is specified these pragmas are ignored. By default, inline expansion is performed.

**/LIBRARY=dir-spec**

Specifies the program library the command works on. The SAS COMPLETE command needs write access to the library. The default library is [.ADALIB].

**/LIST[=file-spec]**

**/NOLIST (D)**

Controls whether a listing file is created.

The default directory for listing files is the current default directory. The default file name is COMPLETE. The default file type is LIS. No wildcard characters are allowed in the file specification. By default, the COMPLETE command does not create a listing file.

**/LOG[=file-spec]**

**/NOLOG (D)**

Controls whether the SAS COMPLETE command writes additional messages onto the specified file. The default file name is SYS\$OUTPUT. The default file type is LOG.

By default, no additional messages are written.

**/MACHINE\_CODE**

**/NOMACHINE\_CODE (D)**

Controls whether a machine code listing is appended to the listing file. /MACHINE\_CODE has no effect if /NOLIST is specified. By default, no machine code listing is appended to the listing file.

**/OPTIMIZE (D)**

**/NOOPTIMIZE**

Controls whether full optimization is applied in generating code. There is no way to specify that only certain optimizations are to be performed.

By default, full optimization is done.

### 4.3 Automatic Compilation

The SYSTEAM Ada System offers three different kinds of automatic compilation. It supports

- automatic recompilation of obsolete units
- automatic compilation of modified sources

In the following the term *recompilation* stands for the recompilation of an obsolete unit using the identical source which was used the last time. (This kind of recompilation could alternatively be implemented by using some appropriate intermediate representation of the obsolete unit.) This definition is stronger than that of the LRM (10.3). If a new version of the source of a unit is compiled we call it *compilation*, not a *recompilation*.

The set of units to be checked for recompilation or new compilation is described by specifying one or more units and the kind of a closure which is to be build on them. In many cases you will simply specify your main program.

The automatic recompilation of obsolete units is supported by the SAS RECOMPILE command. It determines the set of obsolete units and generates a command file for calling the Compiler in an appropriate order. This command file is started by the user either in batch or in interactive mode.

The recompilation is performed using the copy of the obsolete units which is (by default) stored in the library. (If the user does not want to hold a copy of the sources the SAS RECOMPILE command offers the facility to use the original source.)

The automatic compilation of modified sources is supported by the SAS AUTOCOMPILE command. It determines the set of modified sources and generates a command file for calling the Compiler in an appropriate order. This command file is started by the user either in batch or in interactive mode. The basis of both the SAS RECOMPILE and the SAS AUTOCOMPILE command is the information in the library about the dependencies of the concerned units. Thus neither of these commands can handle the compilation of units which have not yet been entered in the library.

The automatic compilation of new sources is supported by the SAS COMPILE command together with the /ANALYZE\_DEPENDENCY qualifier. This command is able to accept a set of sources in any order. It makes a syntactical analysis of the sources

and determines the dependencies. The units "compiled" with this command are entered into the library, but only their names, their dependencies on other units and the name of the source files are stored in the library. Units which are entered this way can be automatically compiled using the SAS AUTOCOMPILE command. They *cannot* be recompiled using the SAS RECOMPILE command because the SAS RECOMPILE command only recompiles units which were already compiled.

The next sections explain the usage of the SAS RECOMPILE command, the SAS AUTOCOMPILE command, and the SAS COMPILE/ANALYZE\_DEPENDENCY command.

### 4.3.1 Recompiling Obsolete Units

The SAS RECOMPILE command supports the automatic recompilation of units which became obsolete because of the (re)compilation of units they depend on. The command gets as a parameter a set of units which are to be used to form the closure of units to be recompiled. The kind of the closure can be specified. The SAS RECOMPILE command generates a command file with a sequence of SAS COMPILE commands to recompile the obsolete units which belong to the computed closure. This command file is started by the user either in batch or in interactive mode. The name of the command file can be specified using the /OUTPUT qualifier.

SAS RECOMPILE	Command Description
<b>Format</b>	
\$ SAS RECOMPILE unit[....]	
<b>Command Qualifiers</b>	<b>Defaults</b>
/[NO]BODIES_ONLY	/NOBODIES_ONLY
/[NO]CHECK	see Text
/[NO]CLOSURE=option	/CLOSURE=EXECUTE
/[NO]CONDITIONAL	/CONDITIONAL
/[NO]INLINE	see Text
/LIBRARY=directory-spec	/LIBRARY=[.ADALIB]
/[NO]LIST[=file-spec]	/NOLIST
/[NO]LOG[=file-spec]	/NOLOG
/[NO]MACHINE_CODE	/NOMACHINE_CODE
/[NO]OPTIMIZE	see Text
/OUTPUT=file-spec	/OUTPUT=[]RECOMPILE.COM
/[NO]SAME	/SAME
<b>Positional Qualifiers</b>	<b>Defaults</b>
/BODY	see Text

## Command Parameters

**unit**

Specifies the unit(s) whose closure is to be built.

## Description

The SAS RECOMPILE command determines the specified closure based on the specified unit(s). Out of the units of the closure it determines the set of units which are obsolete. It generates a command file containing an SAS COMPILE/RECOMPILE command for every obsolete unit. They are compiled in an order consistent with the WITH dependencies and the "body-of" and "subunit-of" dependencies as required by the LRM(10.3).

The SAS RECOMPILE commands uses the copy of the source which is stored in the library for the recompilation. By default, the SAS COMPILE command stores a copy of the source in the library. If there is no copy in the library - because the unit was compiled using the /NOCOPY\_SOURCE qualifier - the SAS RECOMPILE issues a warning and generates an SAS COMPILE command for the original source file name. This name does not include a version number. It is *not* checked whether such a file still exists. This command only performs a real recompilation if the current source is the same which was last compiled.

In the command file each recompilation of a unit is executed under the condition that the recompilation of other units it depends on was successful. Thus useless recompilations are avoided. The generated command file only works correctly if the library was not modified since the command file was generated.

**Note:** If a unit from a parent library is obsolete it is compiled in the sublibrary in which the SAS RECOMPILE command is used. In this case a later recompilation in the parent library may be hidden afterwards.

## Command Qualifiers

**/BODIES\_ONLY**

**/NOBODIES\_ONLY (D)**

Controls whether all units of the closure are recompiled (default) or only the secondary units. This qualifier is only effective if /NOCONDITIONAL is specified.

**/CHECK**

**/NOCHECK**

This qualifier is included in the generated command file and thus affects the generated SAS COMPILE command. See the same qualifier with the SAS COMPILE command.

**/CLOSURE=(EXECUTE|COMPILE)**  
**/NOCLOSURE**

Controls the kind of the closure which is built and which is the basis for the investigation for obsolete units. **/NOCLOSURE** means that only the specified units are checked. **/CLOSURE=COMPILE** means that only those units on which the specified unit(s) transitively depend(s) are regarded. **/CLOSURE=EXECUTE** means that - in addition - all related secondary units and the units they depend on are regarded.

By default, the execution closure is built.

**/CONDITIONAL (D)**  
**/NOCONDITIONAL**

Controls whether only obsolete units are recompiled (default). **/NOCONDITIONAL** means that all units in the closure are recompiled whether they are obsolete or not. This qualifier is useful for recompiling the complete closure with different qualifiers than the last time.

**/INLINE**  
**/NOINLINE**

This qualifier is included in the generated command file and thus affects the generated SAS COMPILE command. See the same qualifier with the SAS COMPILE command.

**/LIBRARY=directory-spec**  
Specifies the program library the command works on.  
The default is [.ADALIB].

**/LIST[=file-spec]**  
**/NOLIST (D)**

This qualifier is included in the generated command file and thus affects the generated SAS COMPILE command. See the same qualifier with the SAS COMPILE command.

**/LOG[=file-spec]**  
**/NOLOG (D)**

This qualifier is included in the generated command file and thus affects the generated SAS COMPILE command. See the same qualifier with the SAS COMPILE command.

**/MACHINE\_CODE**  
**/NOMACHINE\_CODE (D)**

This qualifier is included in the generated command file and thus affects the generated SAS COMPILE command. See the same qualifier with the SAS COMPILE command.

**/OPTIMIZE**

**/NOOPTIMIZE**

This qualifier is included in the generated command file and thus affects the generated SAS COMPILE command. See the same qualifier with the SAS COMPILE command.

**/OUTPUT=file-spec**

Specifies the name of the generated command file. The default directory is []. The default file name is RECOMPILE. The default file type is COM.

**/SAME (D)**

**/NOSAME**

Controls whether the SAS RECOMPILE command inserts certain additional qualifiers in each SAS COMPILE command which is generated.

/SAME means that the same qualifiers for /CHECK, /INLINE and /OPTIMIZE are included which were in effect at the last compilation, unless they are specified explicitly with the SAS RECOMPILE command. By default, additional qualifiers for /CHECK, /INLINE and /OPTIMIZE are inserted in order to do the compilation with the same effect as last time.

### Positional Qualifiers

**/BODY**

specifies that unit stands for the secondary unit with that name. By default, unit denotes the library unit. If unit specifies a subunit, the /BODY qualifier need not be specified.

---

### End of Command Description

---

#### 4.3.2 Compiling New Sources

The SAS AUTOCOMPILE command supports the automatic compilation of units for which a new source exists. The command receives as parameters a set of units which are to be used to form the closure of units to be processed. The kind of closure can be specified. For every unit in the closure, the SAS AUTOCOMPILE checks whether there exists a newer source than that which was used for the last compilation. It generates a command file with a sequence of SAS COMPILE commands to compile the units for which a newer source exists. If a unit to be compiled depends on another unit which is obsolete or which will become obsolete and for which no newer source exists, the SAS AUTOCOMPILE command always adds an appropriate SAS



COMPILE/RECOMPILE command to make it current; the /RECOMPILE qualifier controls which other obsolete units are recompiled, and can indeed be used to specify that the same recompilations are done as if the SAS RECOMPILE command was applied subsequently. The generated command file is started by the user either in batch or in interactive mode. The name of the command file can be specified using the /OUTPUT qualifier.

---

## SAS AUTOCOMPILE

---

## Command Description

---

### Format

\$ SAS AUTOCOMPILE unit[,...]

#### Command Qualifiers

/[NO]BODIES\_ONLY  
/[NO]CHECK  
/[NO]CLOSURE=option  
/[NO]CONDITIONAL  
/[NO]COPY\_SOURCE  
/[NO]INLINE  
/LIBRARY=directory-spec  
/[NO]LIST[=file-spec]  
/[NO]LOG[=file-spec]  
/[NO]MACHINE\_CODE  
/[NO]OPTIMIZE  
/OUTPUT=file-spec  
/[NO]RECOMPILE=option  
/[NO]SAME

#### Defaults

/NOBODIES\_ONLY  
see Text  
/CLOSURE=EXECUTE  
/CONDITIONAL  
/COPY\_SOURCE  
see Text  
/LIBRARY=[.ADALIB]  
/NOLIST  
/NOLOG  
/NOMACHINE\_CODE  
see Text  
/OUTPUT=[]AUTOCOMPILE.COM  
/NORECOMPILE  
/SAME

#### Positional Qualifiers

/BODY

#### Defaults

see Text

### Command Parameters

#### unit

Specifies the unit(s) whose closure is to be built.

### Description

The SAS AUTOCOMPILE command determines the specified closure based on the specified unit(s). It determines the set of units for which a new source exists. The decision is based on the full file name which is stored in the library together with the modification date. If the newest version of the file has a newer modification date than the modification date which is stored in the library then the unit is said to be "new". Units

which were entered with SAS COMPILE/ANALYZE\_DEPENDENCY are always said to be new.

The SAS AUTOCOMPILE command generates a command file containing an SAS COMPILE command for every "new" unit. They are compiled in an order according to the WITH dependencies and the "body-of" and "subunit-of" relations. It is assumed that the dependencies do not change in the new sources.

Inline dependencies are not fully considered by the SAS AUTOCOMPILE command. The SAS AUTOCOMPILE command detects that a unit which is currently current will become obsolete because it depends on another unit (because of an inline call) which will be (re)compiled. The SAS AUTOCOMPILE command does *not* detect that a unit *u* which is not current and will be (re)compiled *will* become dependent on another unit *v* (because of an inline call) which is currently current and will be (re)compiled too but after the compilation of *u*. In this case *u* will become obsolete again when *v* is (re)compiled.

When determining the compilation order the SAS AUTOCOMPILE command tries to choose the same order as last time by considering the compilation dates in the library, where possible. This strategy should solve the inline problem in most cases.

In the generated command file each compilation of a unit is executed under the condition that the compilations of other units it depends on were successful. Thus useless compilations are avoided. The generated command file only works correctly if the library has not been modified since the command file was generated.

The SAS AUTOCOMPILE command does not fully handle the problem which arises when several compilation units are contained within one source file; it only avoids the multiple compilation of the same source file. If you want to use the SAS AUTOCOMPILE command it is recommended not to keep several compilation units in one source.

### Command Qualifiers

/BODIES\_ONLY

/NOBODIES\_ONLY (D)

Controls whether all new units of the closure are compiled (default) or only the secondary units. This qualifier is only effective if /NOCONDITIONAL is specified.

/CHECK

/NOCHECK

This qualifier is included in the generated command file and thus affects the generated SAS COMPILE command. See the same qualifier with the SAS COMPILE command.

**/CLOSURE=(EXECUTE|COMPILE)**

**/NOCLOSURE**

Controls the kind of the closure which is built and which is the basis for the investigation for new sources. **/NOCLOSURE** means that only the specified units are checked. **/CLOSURE=COMPILE** means that only those units on which the specified unit(s) transitively depend(s) are regarded. **/CLOSURE=EXECUTE** means that - in addition - all related secondary units and the units they depend on are regarded.

By default, the execution closure is investigated for new sources.

**/CONDITIONAL (D)**

**/NOCONDITIONAL**

Controls whether the check for new sources is performed (default). **/NOCONDITIONAL** means that all units in the closure are compiled disregarding the modification date. This qualifier is useful for compiling the complete closure with different qualifiers than the last time.

**/COPY\_SOURCE (D)**

**/NOCOPY\_SOURCE**

This qualifier is included in the generated command file and thus affects the generated SAS COMPILE command. See the same qualifier with the SAS COMPILE command. This qualifier has no effect for the recompilation of obsolete units in accordance with the SAS RECOMPILE command where **/COPY\_SOURCE** cannot be specified.

**/INLINE**

**/NOINLINE**

This qualifier is included in the generated command file and thus affects the generated SAS COMPILE command. See the same qualifier with the SAS COMPILE command.

**/LIBRARY=directory-spec**

Specifies the program library the command works on. The SAS AUTO-COMPILE command needs read access to the library. For executing the generated command file you need write access. The default is [.ADALIB].

**/LIST[=file-spec]**

**/NOLIST (D)**

This qualifier is included in the generated command file and thus affects the generated SAS COMPILE command. See the same qualifier with the SAS COMPILE command.

**/LOG[=file-spec]**

**/NOLOG (D)**

This qualifier is included in the generated command file and thus affects the generated SAS COMPILE command. See the same qualifier with the SAS COMPILE command.

**/MACHINE\_CODE**

**/NOMACHINE\_CODE (D)**

This qualifier is included in the generated command file and thus affects the generated SAS COMPILE command. See the same qualifier with the SAS COMPILE command.

**/OPTIMIZE**

**/NOOPTIMIZE**

This qualifier is included in the generated command file and thus affects the generated SAS COMPILE command. See the same qualifier with the SAS COMPILE command.

**/OUTPUT=file-spec**

Specifies the name of the generated command file. The default directory is []. The default file name is AUTOCOMPILE. The default file type is COM.

**/RECOMPILE=(OBSOLETE|ALL)**

**/NORECOMPILE (D)**

Controls whether the SAS AUTOCOMPILE command additionally recompiles obsolete units. With /NORECOMPILE only those units are recompiled which are obsolete or become obsolete *and* are used by other units which are to be compiled because of new sources. /RECOMPILE=OBSOLETE additionally recompiles those units of the considered closure which will become obsolete during the compilation of new sources. This option specifies that there shall not be more obsolete units after the execution of the command file than before. /RECOMPILE=ALL specifies that all obsolete units of the closure and all units which will become obsolete are recompiled. This is equivalent to a subsequent call of the SAS RECOMPILE command after the run of the command file generated by the SAS AUTOCOMPILE command.

**/SAME (D)**

**/NOSAME**

Controls whether the SAS AUTOCOMPILE command inserts certain additional qualifiers in each SAS COMPILE command which is generated. /SAME means that the same qualifiers for /CHECK, /INLINE and /OPTIMIZE are included which were in effect at the last compilation, unless they are specified explicitly with the SAS AUTOCOMPILE command. By default, additional qualifiers for /CHECK, /INLINE and /OPTIMIZE are inserted in order to do the compilation with the same effect as last time.

## Positional Qualifiers

**/BODY**

specifies that unit stands for the secondary unit with that name. By default, unit denotes the library unit. If unit specifies a subunit, the /BODY qualifier need not be specified.

---

**End of Command Description**

---

**4.3.3 First compilation**

The SYSTEAM Ada System supports the first compilation of sources for which no compilation order is known by the SAS COMPILE/ANALYZE\_DEPENDENCY command in combination with the SAS AUTOCOMPILE command.

With the /ANALYZE\_DEPENDENCY qualifier the Compiler accepts sources in any order and performs the syntax analysis. If the sources are syntactically correct the units which are defined by the sources are entered into the library. Their names, their dependencies on other units and the name of the source files are stored in the library. Units which are entered this way can be automatically compiled using the SAS AUTOCOMPILE command, i.e. the Autocompiler computes the first compilation order for the new sources. The name of the main program, of course, must be known and specified with the SAS AUTOCOMPILE command.

Note that the SAS COMPILE/ANALYZE\_DEPENDENCY command replaces other units in the library with the same name as a new one. Thus the library may be modified even if the new units contain semantic errors; but the errors will not be detected until the command file generated by the SAS AUTOCOMPILE command is run. Hence it is recommended to use an empty sublibrary if you do not know anything about the set of new sources.

Wild cards in the SAS COMPILE command are intended to be used together with the /ANALYZE\_DEPENDENCY qualifier.

If there are several sources containing units with the same name the last analyzed one will be kept in the library.

The SAS AUTOCOMPILE command issues special warnings if the information about the new units is incomplete or inconsistent.

## 4.4 Compiler Listing

By default, messages of the Compiler are listed on SYS\$OUTPUT. A complete listing can be obtained on a file by using the /LIST qualifier with the SAS COMPILE, SAS COMPLETE or SAS LINK command. The generated listing file(s) will contain the whole source together with the messages of the Compiler/Completer.

The listing for a compilation unit starts with the kind and the name of the current unit.

*Example:*

```
=  PROCEDURE  MAIN
```

The format effectors ASCII.HT, ASCII.VT, ASCII.CR, ASCII.LF and ASCII.FF are represented by a ''' character in the listing. In any case, those source lines which are included in the listing are numbered to make locating them in the source file easy.

Errors are classified into SYMBOL ERROR, SYNTAX ERROR, SEMANTIC ERROR, RESTRICTION, COMPILER ERROR, WARNING and INFORMATION:

### SYMBOL ERROR

pinpoints an inappropriate lexical element. "Inappropriate" can mean "inappropriate in the given context". For example, '2' is a lexical element of Ada, but it is not appropriate in the literal 2#012#.

### SYNTAX ERROR

indicates a violation of the Ada syntax rules as given in the LRM(Appendix E).

### SEMANTIC ERROR

indicates a violation of Ada language rules other than the syntax rules.

### RESTRICTION

indicates a restriction of this implementation. Examples are representation clauses which are provided by the language but are not supported in this implementation; or situations in which the internal storage capacity of the Compiler for some sort of entity is exceeded.

### COMPILER ERROR

We hope you will never see a message of this sort.

### WARNING

messages tell the user facts which are likely to cause errors (for example, the raising of exceptions) at runtime.

### INFORMATION

messages tell the user facts which may be useful to know but probably do not endanger the correct running of the program. Examples are that a library unit named in a context clause is not used in the current compilation unit, or that another unit (which names the current compilation unit in a context clause) is made obsolete by the current compilation.

Warnings and information messages have no influence on the success of a compilation. If there are any other diagnostic messages, the compilation was unsuccessful.

All error messages are self-explanatory. If a source line contains errors, the error messages for that source line are printed immediately below it. The exact position in the source to which an error message refers is marked by a number. This number is also used to relate different error messages given for one line to their respective source positions.

In order to enable semantic analysis to be carried out even if a program is syntactically incorrect, the Compiler corrects syntax errors automatically by inserting or deleting symbols. The source positions of insertions/deletions are marked with a vertical bar and a number. The number has the same meaning as above. If a larger region of the source text is affected by a syntax correction, this region is located for the user by repeating the number and the vertical bar at the end as well, with dots in between these bracketing markings.

A complete Compiler listing follows which shows the most common kinds of error messages, the technique for marking affected regions and the numbering scheme for relating error messages to source positions. It is slightly modified so that it fits into the page width of this document:

```
*****
**                                                                 **
**  SYSTEAM ADA - COMPILER          VAX/VMS  1.82                **
**                                                                 **
**  90-01-29/08:39:44                                             **
**                                                                 **
*****

=====
=
=                               Started at   : 08:39:44           =
=
=
=  PROCEDURE  LISTING_EXAMPLE                                     =
=
    1      PROCEDURE listing-example IS
    2      abc : procedure integer RANGE 0 .. 9 := 10E-1;
              |1.....1|
                                           1

>>>>> SYNTAX ERROR
        Symbol(s)  deleted (1)
>>>>> SYMBOL ERROR (1)   An exponent for an integer literal must not
```

## APPENDIX C

### APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. The package STANDARD is included in the implementer's Chapter 13, Predefined Language Environment, on pages 275ff.



## 13 Predefined Language Environment

The predefined language environment comprises the package standard, the language-defined library units and the implementation-defined library units.

### 13.1 The Package STANDARD

The specification of the package standard is outlined here; it contains all predefined identifiers of the implementation.

PACKAGE standard IS

    TYPE boolean IS (false, true);

    -- The predefined relational operators for this type are as follows:

    -- FUNCTION "=" (left, right : boolean) RETURN boolean;  
    -- FUNCTION "/=" (left, right : boolean) RETURN boolean;  
    -- FUNCTION "<" (left, right : boolean) RETURN boolean;  
    -- FUNCTION "<=" (left, right : boolean) RETURN boolean;  
    -- FUNCTION ">" (left, right : boolean) RETURN boolean;  
    -- FUNCTION ">=" (left, right : boolean) RETURN boolean;

    -- The predefined logical operators and the predefined logical  
    -- negation operator are as follows:

    -- FUNCTION "AND" (left, right : boolean) RETURN boolean;  
    -- FUNCTION "OR" (left, right : boolean) RETURN boolean;  
    -- FUNCTION "XOR" (left, right : boolean) RETURN boolean;  
  
    -- FUNCTION "NOT" (right : boolean) RETURN boolean;

    -- The universal type universal\_integer is predefined.

    TYPE integer IS RANGE - 2\_147\_483\_648 .. 2\_147\_483\_647;

    -- The predefined operators for this type are as follows:

    -- FUNCTION "=" (left, right : integer) RETURN boolean;  
    -- FUNCTION "/=" (left, right : integer) RETURN boolean;

```

-- FUNCTION "<" (left, right : integer) RETURN boolean;
-- FUNCTION "<=" (left, right : integer) RETURN boolean;
-- FUNCTION ">" (left, right : integer) RETURN boolean;
-- FUNCTION ">=" (left, right : integer) RETURN boolean;

-- FUNCTION "+" (right : integer) RETURN integer;
-- FUNCTION "-" (right : integer) RETURN integer;
-- FUNCTION "ABS" (right : integer) RETURN integer;

-- FUNCTION "+" (left, right : integer) RETURN integer;
-- FUNCTION "-" (left, right : integer) RETURN integer;
-- FUNCTION "*" (left, right : integer) RETURN integer;
-- FUNCTION "/" (left, right : integer) RETURN integer;
-- FUNCTION "REM" (left, right : integer) RETURN integer;
-- FUNCTION "MOD" (left, right : integer) RETURN integer;

-- FUNCTION "***" (left : integer; right : integer) RETURN integer;

-- An implementation may provide additional predefined integer types.
-- It is recommended that the names of such additional types end
-- with INTEGER as in SHORT_INTEGER or LONG_INTEGER. The
-- specification of each operator for the type universal_integer, or
-- for any additional predefined integer type, is obtained by
-- replacing INTEGER by the name of the type in the specification
-- of the corresponding operator of the type INTEGER, except for the
-- right operand of the exponentiating operator.

TYPE short_integer IS RANGE - 32_768 .. 32_767;

TYPE short_short_integer IS RANGE - 128 .. 127;

-- The universal type universal_real is predefined.

TYPE float IS DIGITS 9 RANGE
    - 16#0.7FFF_FFFF_FFFF_FF8#E+32 ..
    16#0.7FFF_FFFF_FFFF_FF8#E+32;
-- the corresponding machine type is D-FLOAT

-- The predefined operators for this type are as follows:

-- FUNCTION "=" (left, right : float) RETURN boolean;
-- FUNCTION "/=" (left, right : float) RETURN boolean;
-- FUNCTION "<" (left, right : float) RETURN boolean;
-- FUNCTION "<=" (left, right : float) RETURN boolean;
-- FUNCTION ">" (left, right : float) RETURN boolean;
-- FUNCTION ">=" (left, right : float) RETURN boolean;

```

```

-- FUNCTION "+" (right : float) RETURN float;
-- FUNCTION "-" (right : float) RETURN float;
-- FUNCTION "ABS" (right : float) RETURN float;

-- FUNCTION "+" (left, right : float) RETURN float;
-- FUNCTION "-" (left, right : float) RETURN float;
-- FUNCTION "*" (left, right : float) RETURN float;
-- FUNCTION "/" (left, right : float) RETURN float;

-- FUNCTION "***" (left : float; right : integer) RETURN float;

-- An implementation may provide additional predefined floating
-- point types. It is recommended that the names of such additional
-- types end with FLOAT as in SHORT_FLOAT or LONG_FLOAT.
-- The specification of each operator for the type universal_real,
-- or for any additional predefined floating point type, is obtained
-- by replacing FLOAT by the name of the type in the specification of
-- the corresponding operator of the type FLOAT.

TYPE short_float IS DIGITS 6 RANGE
    - 16#0.7FFF_FF8#E+32 .. 16#0.7FFF_FF8#E+32;
-- the corresponding machine type is F-FLOAT

TYPE long_float IS DIGITS 15 RANGE
    - 16#0.7FFF_FFFF_FFFF_FC#E+256 ..
      16#0.7FFF_FFFF_FFFF_FC#E+256;
-- the corresponding machine type is G-FLOAT

TYPE long_long_float IS DIGITS 33 RANGE
    - 16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_C#E+4096 ..
      16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_C#E+4096;
-- the corresponding machine type is H-FLOAT

-- In addition, the following operators are predefined for universal
-- types:

-- FUNCTION "*" (left : UNIVERSAL_INTEGER; right : UNIVERSAL_REAL)
-- RETURN UNIVERSAL_REAL;
-- FUNCTION "*" (left : UNIVERSAL_REAL; right : UNIVERSAL_INTEGER)
-- RETURN UNIVERSAL_REAL;
-- FUNCTION "/" (left : UNIVERSAL_REAL; right : UNIVERSAL_INTEGER)
-- RETURN UNIVERSAL_REAL;

-- The type universal_fixed is predefined.
-- The only operators declared for this type are

-- FUNCTION "*" (left : ANY_FIXED_POINT_TYPE;

```

```

right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;
-- FUNCTION "/" (left : ANY_FIXED_POINT_TYPE;
right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;

```

```

-- The following characters form the standard ASCII character set.
-- Character literals corresponding to control characters are not
-- identifiers.

```

TYPE character IS

```

(nul,  soh,  stx,  etx,      eot,  enq,  ack,  bel,
 bs,   ht,   lf,   vt,      ff,   cr,   so,   si,
 dle,  dc1,  dc2,  dc3,      dc4,  nak,  syn,  etb,
 can,  em,   sub,  esc,      fs,   gs,   rs,   us,
 ' ',  '!',  '"',  '#',      '$',  '%',  '&',  '...',
 '(',  ')',  '*',  '+',      ',',  '-',  '.',  '/',
 '0',  '1',  '2',  '3',      '4',  '5',  '6',  '7',
 '8',  '9',  ':',  ';',      '<',  '=',  '>',  '?',
 '@',  'A',  'B',  'C',      'D',  'E',  'F',  'G',
 'H',  'I',  'J',  'K',      'L',  'M',  'N',  'O',
 'P',  'Q',  'R',  'S',      'T',  'U',  'V',  'W',
 'X',  'Y',  'Z',  '[',      '\',  ']',  '^',  '_',
 '...', 'a',  'b',  'c',      'd',  'e',  'f',  'g',
 'h',  'i',  'j',  'k',      'l',  'm',  'n',  'o',
 'p',  'q',  'r',  's',      't',  'u',  'v',  'w',
 'x',  'y',  'z',  '{',      '|',  '}',  '~',  del);

```

```

FOR character USE -- 128 ascii CHARACTER SET WITHOUT HOLES
(0, 1, 2, 3, 4, 5, ..., 125, 126, 127);

```

```

-- The predefined operators for the type CHARACTER are the same as
-- for any enumeration type.

```

PACKAGE ascii IS

-- Control characters:

```

nul : CONSTANT character := nul;    soh : CONSTANT character := soh;
stx : CONSTANT character := stx;    etx : CONSTANT character := etx;
eot : CONSTANT character := eot;    enq : CONSTANT character := enq;
ack : CONSTANT character := ack;    bel : CONSTANT character := bel;
bs  : CONSTANT character := bs;     ht  : CONSTANT character := ht;
lf  : CONSTANT character := lf;     vt  : CONSTANT character := vt;
ff  : CONSTANT character := ff;     cr  : CONSTANT character := cr;
so  : CONSTANT character := so;     si  : CONSTANT character := si;
dle : CONSTANT character := dle;    dc1 : CONSTANT character := dc1;
dc2 : CONSTANT character := dc2;    dc3 : CONSTANT character := dc3;
dc4 : CONSTANT character := dc4;    nak : CONSTANT character := nak;
syn : CONSTANT character := syn;    etb : CONSTANT character := etb;
can : CONSTANT character := can;    em  : CONSTANT character := em;

```

```
sub : CONSTANT character := sub;   esc : CONSTANT character := esc;
fs  : CONSTANT character := fs;    gs  : CONSTANT character := gs;
rs  : CONSTANT character := rs;    us  : CONSTANT character := us;
del : CONSTANT character := del;
```

-- Other characters:

```
exclam    : CONSTANT character := '!';
quotation : CONSTANT character := '"';
sharp     : CONSTANT character := '#';
dollar    : CONSTANT character := '$';
percent   : CONSTANT character := '%';
ampersand : CONSTANT character := '&';
colon     : CONSTANT character := ':';
semicolon : CONSTANT character := ';';
query     : CONSTANT character := '?';
at_sign   : CONSTANT character := '@';
l_bracket : CONSTANT character := '[';
back_slash : CONSTANT character := '\';
r_bracket : CONSTANT character := ']';
circumflex : CONSTANT character := '^';
underline : CONSTANT character := '_';
grave     : CONSTANT character := '`';
l_brace   : CONSTANT character := '{';
bar       : CONSTANT character := '|';
r_brace   : CONSTANT character := '}';
tilde     : CONSTANT character := '~';
```

```
lc_a : CONSTANT character := 'a';
...
lc_z : CONSTANT character := 'z';
```

END ascii;

-- Predefined subtypes:

```
SUBTYPE natural IS integer RANGE 0 .. integer'last;
SUBTYPE positive IS integer RANGE 1 .. integer'last;
```

-- Predefined string type:

```
TYPE string IS ARRAY(positive RANGE <>) OF character;
```

```
PRAGMA pack(string);
```

-- The predefined operators for this type are as follows:

```

-- FUNCTION "=" (left, right : string) RETURN boolean;
-- FUNCTION "/" (left, right : string) RETURN boolean;
-- FUNCTION "<" (left, right : string) RETURN boolean;
-- FUNCTION "<=" (left, right : string) RETURN boolean;
-- FUNCTION ">" (left, right : string) RETURN boolean;
-- FUNCTION ">=" (left, right : string) RETURN boolean;

-- FUNCTION "&" (left : string;    right : string)    RETURN string;
-- FUNCTION "&" (left : character; right : string)   RETURN string;
-- FUNCTION "&" (left : string;    right : character) RETURN string;
-- FUNCTION "&" (left : character; right : character) RETURN string;

```

```

TYPE duration IS DELTA 2#1.0#E-14 RANGE
    - 131_072.0 .. 131_071.999_938_964_843_75;

```

```

-- The predefined operators for the type DURATION are the same
-- as for any fixed point type.

```

```

-- the predefined exceptions:

```

```

constraint_error : EXCEPTION;
numeric_error    : EXCEPTION;
program_error    : EXCEPTION;
storage_error    : EXCEPTION;
tasking_error    : EXCEPTION;

```

```

END standard;

```

## 13.2 Language-Defined Library Units

The following language-defined library units are included in the master library:

```

The package system
The package calendar
The generic procedure unchecked_deallocation
The generic function unchecked_conversion
The package io_exceptions
The generic package sequential_io
The generic package direct_io
The package text_io
The package low_level_io

```

### 13.3 Implementation-Defined Library Units

The master library also contains the implementation-defined library units `collection_manager` and `timing`.

#### 13.3.1 The Package `COLLECTION_MANAGER`

In addition to unchecked storage deallocation (cf. LRM(§13.10.1)), this implementation provides the generic package `collection_manager`, which has advantages over unchecked deallocation in some applications; e.g. it makes it possible to clear a collection with a single reset operation. See §15.10 for further information on the use of the collection manager and unchecked deallocation.

The package specification is:

GENERIC

    TYPE `elem` IS LIMITED PRIVATE;

    TYPE `acc` IS ACCESS `elem`;

PACKAGE `collection_manager` IS

    TYPE `status` IS LIMITED PRIVATE;

    PROCEDURE `mark` (`s` : OUT `status`);

        -- Marks the heap of type ACC and

        -- delivers the actual status of this heap.

    PROCEDURE `release` (`s` : IN `status`);

        -- Restore the status `s` on the collection of ACC.

        -- RELEASE without previous MARK raises `CONSTRAINT_ERROR`

    PROCEDURE `reset`;

        -- Deallocate all objects on the heap of ACC

PRIVATE

    -- private declarations

```
END collection_manager;
```

A call of the procedure `release` with an actual parameter `s` causes the storage occupied by those objects of type `acc` which were allocated after the call of `mark` that delivered `s` as result, to be reclaimed. A call of `reset` causes the storage occupied by all objects of type `acc` which have been allocated so far to be reclaimed and cancels the effect of all previous calls of `mark`.

See §15.2.1 for information on static and dynamic collections and the attribute `STORAGE_SIZE`.

### 13.3.2 The Package `TIMING`

The package `timing` provides a facility for CPU-time measurement. The package specification is:

`PACKAGE timing IS`

```
    FUNCTION cpu_time RETURN natural;
```

```
    timing_error : EXCEPTION;
```

```
END timing;
```

A call of the function `cpu_time` returns the CPU-time consumed by the running process in milliseconds. The value `natural'last` will be reached after 24 days, 20 hours, 31 minutes, 23 seconds and 647 milliseconds.

The exception `timing_error` will be raised if a `constraint_error` or `numeric_error` occurs within `cpu_time`.



## 15 Appendix F

This chapter, together with the Chapters 16 and 17, is the Appendix F required in the LRM, in which all implementation-dependent characteristics of an Ada implementation are described.

### 15.1 Implementation-Dependent Pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

#### 15.1.1 Predefined Language Pragmas

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here.

##### CONTROLLED

has no effect.

##### ELABORATE

is fully implemented. The SYSTEAM Ada System assumes a PRAGMA elaborate, i.e. stores a unit in the library as if a PRAGMA elaborate for a unit *u* was given, if the compiled unit contains an instantiation of *u* (or for a generic program unit in *u*) and if it is clear that *u* *must* have been elaborated before the compiled unit. In this case an appropriate information message is given. By this means it is avoided that an elaboration order is chosen which would lead to a PROGRAM\_ERROR when elaborating the instantiation.

##### INLINE

Inline expansion of subprograms is supported with the following restrictions: the subprogram must not contain declarations of other subprograms, tasks, generic units or body stubs. If the subprogram is called recursively only the outer call of this subprogram will be expanded.

**INTERFACE**

is supported for ASSEMBLER and VMS. `PRAGMA interface( assembler,... )` provides an interface with the internal calling conventions of the SYSTEAM Ada System. See §15.1.3 for further description.

`PRAGMA interface(VMS,...)` is provided to support the VAX procedure calling standard. §15.1.4 describes how to use this pragma.

`PRAGMA interface` should always be used in connection with the `PRAGMA external_name` (see §15.1.2), otherwise the Compiler will generate an internal name that leads to an unsolved reference during linking. These generated names are prefixed with an underline; therefore the user should not use names beginning with an underline.

**LIST**

is fully implemented. Note that a listing is only generated when the `/LIST` qualifier is specified with the `SAS COMPILE` (or `SAS COMPLETE` or `SAS LINK`) command.

**MEMORY\_SIZE**

has no effect.

**OPTIMIZE**

has no effect; but see also the `/OPTIMIZER` qualifier with the `SAS COMPILE` command, §4.1

**PACK**

see §16.1.

**PAGE**

is fully implemented. Note that form feed characters in the source do not cause a new page in the listing. They are - as well the other format effectors (horizontal tabulation, vertical tabulation, carriage return, and line feed) - replaced by a ~ character in the listing.

**PRIORITY**

There are two implementation-defined aspects of this pragma: First, the range of

the subtype priority, and second, the effect on scheduling (Chapter 14) of not giving this pragma for a task or main program. The range of subtype priority is 0 .. 15, as declared in the predefined library package system (see §15.3); and the effect on scheduling of leaving the priority of a task or main program undefined by not giving PRAGMA priority for it is the same as if the PRAGMA priority 0 had been given (i.e. the task has the lowest priority).

**SHARED**

is fully supported.

**STORAGE\_UNIT**

has no effect.

**SUPPRESS**

has no effect, but see §15.1.2 for the implementation-defined PRAGMA suppress\_all.

**SYSTEM\_NAME**

has no effect.

**15.1.2 Implementation-Defined Pragmas****BYTE\_PACK**

see §16.1.

**EXTERNAL\_NAME (<string>, <ada\_name>)**

<ada\_name> specifies the name of a subprogram or of an object declared in a library package, <string> must be a string literal. It defines the external name of the specified item. The Compiler uses a symbol with this name in the call instruction for the subprogram. The subprogram declaration of <ada\_name> must precede this pragma. If several subprograms with the same name satisfy this requirement the pragma refers to that subprogram which is declared last.

Upper and lower cases are distinguished within <string>, i.e. <string> must be given exactly as it is to be used by external routines. This pragma will be used in

connection with the pragmas interface (vms) or interface (assembler) (see §15.1.1).

### RESIDENT (<ada\_name>)

this pragma causes the value of the object to be held in memory and prevents assignments of a value to the object <ada\_name> from being eliminated by the optimizer (see §4.1) of the SYSTEAM Ada Compiler. The following code sequence demonstrates the intended usage of the pragma:

```

...
x : integer;
a : SYSTEM.address;
...
BEGIN
  x := 5;
  a := x'ADDRESS;
  do_something (a);  -- let do_something be a non-local
                    -- procedure
                    -- a.ALL will be read in the body
                    -- of do_something
  x := 6;
  ...

```

If this code sequence is compiled by the SYSTEAM Ada Compiler with the /OPTIMIZE qualifier the statement `x := 5;` will be eliminated because from the point of view of the optimizer the value of `x` is not used before the next assignment to `x`. Therefore

```
PRAGMA resident (x);
```

should be inserted after the declaration of `x`.

This pragma can be applied to all those kinds of objects for which the address clause is supported (cf. §16.5).

It will often be used in connection with the PRAGMA interface (vms, ... ) (see §15.1.4).

### SUPPRESS\_ALL

causes all the runtime checks described in the LRM (§11.7) to be suppressed; this pragma is only allowed at the start of a compilation before the first compilation unit; it applies to the whole compilation.

### 15.1.3 Pragma Interface (Assembler,...)

This section describes the internal calling conventions of the SYSTEAM Ada System, which are the same ones which are used for subprograms for which a PRAGMA interface (ASSEMBLER,...) is given. Thus the actual meaning of this pragma is simply that the body needs and must not be provided in Ada, but in object form using the /EXTERNAL qualifier with the SAS LINK command.

In many cases it is more convenient to follow the VAX procedure calling standard. Therefore the SYSTEAM Ada System provides the PRAGMA interface(VMS,...), which supports the standard return of the function result and the standard register saving. This pragma is described in the next section.

The internal calling conventions are explained in four steps:

- Parameter passing mechanism
- Ordering of parameters
- Type mapping
- Saving registers

#### *Parameter passing mechanism:*

The parameters of a call to a subprogram are placed by the caller in an area called *parameter block*. This area is aligned on a longword boundary and contains parameter values (for parameter of scalar types), descriptors (for parameter of composite types) and alignment gaps.

For a function subprogram an extra field is assigned at the beginning of the parameter block containing the function result upon return. Thus the return value of a function is treated like an anonymous parameter of mode OUT. No special treatment is required for a function result except for return values of an unconstrained array type (see below).

A subprogram is called using the CALLG instruction. The address pointing to the beginning of the parameter block and the code address of the subprogram are specified as operands. Within the called subprogram the parameter block is accessed via the argument pointer AP.

In general, the ordering of the parameter values within the parameter block does not agree with the order specified in the Ada subprogram specification. When determining the position of a parameter within the parameter block the calling mechanism and the size and alignment requirements of the parameter type are considered. The size and alignment requirements and the passing mechanism are described in the following:

Scalar parameters or parameters of access types are passed by value, i.e. the values of the actual parameters of modes IN or IN OUT are copied into the parameter block

before the call. Then, after the subprogram has returned, values of the actual parameters of modes IN OUT and OUT are copied out of the parameter block into the associated actual parameters. The parameters are aligned within the parameter block according to their size: A parameter with a size of 8, 16 or 32 bits (or a multiple of 8 bits greater than 32) has an alignment of 1, 2 or 4 (which means that the object is aligned to a byte, word or longword boundary within the parameter block). If the size of the parameter is not a multiple of 8 bits (which may be achieved by attaching a size specification to the parameter's type in case of an integer, enumeration or fixed point type) it will be byte aligned. Parameters of access types are always aligned to a longword boundary.

For parameters of composite types, descriptors are placed in the parameter block instead of the complete object values. A descriptor contains the address of the actual parameter object and, possibly, further information dependent on the specific parameter type. The following composite parameter types are distinguished:

- A parameter of a constrained array type is passed by reference for all parameter modes.
- For a parameter of an unconstrained array type, the descriptor consists of the address of the actual array parameter followed by the bounds for each index range in the array (i.e. FIRST(1), LAST(1), FIRST(2), LAST(2), ...). The space allocated for the bound elements in the descriptor depends on the type of the index constraint.
- For functions whose return value is an unconstrained array type a descriptor for the array is passed in the parameter block as for parameters of mode OUT. The fields for its address and all array index bounds are filled up by the function before it returns. In contrast to the procedure for an OUT parameter, the function allocates the array in its own stack space. The function then returns without releasing its stack space. After the function has returned, the calling routine copies the array into its own memory space and then deallocates the stack memory of the function.
- A constrained record parameter is passed by reference for all parameter modes.
- For an unconstrained record parameter of mode IN, the parameter is passed by reference using the address pointing to the record.  
If the parameter has mode OUT or IN OUT, the value of the CONSTRAINED attribute applied to the actual parameter is passed as an additional boolean IN parameter (which occupies one byte in the parameter block and is aligned to a byte boundary). The boolean IN parameter and the address are treated like two consecutive parameters in a subprogram specification, i.e. the positions of the two parameters within the parameter block are determined independently of each other.

For all kinds of composite parameter types the pointer pointing to the actual parameter object is represented by a 32 bit address, which is always aligned to a longword boundary.

*Ordering of parameters:*

The ordering of the parameters in the parameter block is determined as follows:

The parameters are processed in the order they are defined in the Ada subprogram specification. For a function the return value is treated as an anonymous parameter of mode OUT at the start of the parameter list. Because of the size and alignment requirements of a parameter it is not always possible to place parameters in such a way that two consecutive parameters are densely located in the parameter block. In such a situation a gap, i.e. a piece of memory space which is not associated with a parameter, exists between two adjacent parameters. Consequently, the size of the parameter block will be larger than the sum of the sizes used for all parameters. In order to minimize the size of the gaps in a parameter block an attempt is made to fill each gap with a parameter that occurs later in the parameter list. If during the allocation of space within the parameter block a parameter is encountered whose size and alignment fit the characteristics of an available gap, then this gap is allocated for the parameter instead of appending it at the end of the parameter block. As each parameter will be aligned to a byte, word or longword boundary the size of any gap may be one, two or three bytes. Every gap of size three bytes can be treated as two gaps, one of size one byte with an alignment of 1 and one of size two bytes with an alignment of 2. So, if a parameter of size two is to be allocated, a two byte gap, if available, is filled up. A parameter of size one will fill a one byte gap. If none exists but a two byte gap is available, this is used as two one byte gaps. By this first fit algorithm all parameters are processed in the order they occur in the Ada program.

A called subprogram accesses each parameter for reading or writing using the argument pointer AP incremented by an offset from the start of the parameter block suitable for the parameter. So the value of a parameter of a scalar type or an access type is read (or written) directly from (into) the parameter block. For a parameter of a composite type the actual parameter value is accessed via the descriptor stored in the parameter block which contains a pointer to the actual object.

*Type mapping:*

To access individual components of array or record types, knowledge about the type mapping for array and record types is required. An array is stored as a sequential concatenation of all its components. Normally, pad bits are used to fill each component to a byte, word, longword or a multiple thereof in dependence on the size and alignment requirements of the components' subtype. This padding may be influenced using one of the PRAGMAs `pack` or `byte_pack` (cf. §16.1). The offset of an individual array component is then obtained by multiplying the padded size of one array component by the number of components stored in the array before it. This number may be determined from the number of elements for each dimension using the fact that the array elements are stored column by column. (For unconstrained arrays the number of elements for each dimension can be found in the descriptor stored in the parameter block.)

A record object is implemented as a concatenation of its components. Initially, locations are reserved for those components that have a component clause applied to them. Then locations for all other components are reserved. Any gaps large enough to hold components without component clauses are filled, so in general the record components are rearranged. Components in record variants are overlaid. The ordering mechanism of the components within a record is in principle the same as that for ordering the parameters in the parameter block.

A record may hold implementation-dependent components (cf. §16.4). For a record component whose size depends on discriminants, a generated component holds the offset of the record component within the record object. If a record type includes variant parts there may be a generated component (cf. §16.4) holding the size of the record object. This size component is allocated as the first component within the record object if this location is not reserved by a component clause. Since the mapping of record types is rather complex you should introduce record component clauses for each record component if you want to pass an object of that type to a non Ada subprogram to be sure to access the components correctly.

#### *Saving registers:*

The last aspect of the calling conventions discussed here is that of saving registers. The calling subprogram assumes that the values of the registers R0 .. R5 will be destroyed by the called subprogram and saves them of its own accord. If the called subprogram wants to modify further register these must be specified in the procedure entry mask to ensure that they are pushed on the stack when entering the subprogram and restored when leaving it. This differs from the VAX procedure calling standard, which demands that all registers which are modified by the subprogram except R0 and R1 must be specified in the mask.

#### **15.1.4 Pragma Interface(VMS,...)**

The SYSTEAM Ada System supports `PRAGMA interface(VMS,...)`.

With the help of this pragma *and* by obeying some rules (described below) subprograms can be called which follow the VAX procedure calling standard. As the user must know something about the internal calling conventions of the SYSTEAM Ada System we recommend reading §15.1.3 before reading this section and before using `PRAGMA interface(VMS,...)`.

In comparison to `PRAGMA interface(assembler,...)` `PRAGMA interface(VMS,...)` has two additional effects:



- If the subprogram is a function it moves the function result in R0 (or R0 and R1 for results of size 8 byte) into the allocated area at the beginning of the parameter block.
- It controls register allocation in such a way that it is assumed (according to the standard) that the called subprogram only modifies the registers R0 and R1.

The code for subprograms for which `PRAGMA Interface(VMS,...)` is specified can reside in one of the VMS object libraries, which are searched by default, or can be provided using the `/EXTERNAL` qualifier with the SAS LINK command.

In order to follow the VAX procedure calling standard in all aspects the user has to cover three other aspects:

- The order of the parameters in the parameter block must be the same as in the Ada procedure specification. Because of the gap optimization - as described in the previous section - you can enforce this by using only parameters of 4 byte length (for example integer or access types). According to the VAX procedure calling standard shorter parameters are not used anyway. Note, that the Compiler will use a shorter representation where possible, even if there is no PRAGMA pack and no a representation clause. Thus a parameter of type boolean cannot be passed by value; you must specify integer instead, and pass the appropriate corresponding integer value (0 or 1). Most of the parameters of the VMS system calls or of the run time library are passed by reference anyway. In this case the parameter can be declared to be of type `SYSTEM.ADDRESS` and the actual parameter is then `<variable>'ADDRESS`. For the function result the same rules apply as for the parameters, with one exception: You can also use the (predefined) types `FLOAT` and `LONG_FLOAT`, which both have a size of 8 bytes (or equivalent floating point types which are mapped onto these two predefined types.)
- It is an additional convention that the first parameter in the parameter block is of type integer and denotes the number of parameters which are passed. This is easily achieved by the user by adding a first parameter of type integer with a default value. By this means, the calls are not affected, provided parameter association is always named and not positional.

Note: `PRAGMA interface(vms,...)` adds an additional parameter in front of this if the subprogram is a function. The Compiler then correctly passes the address in the parameter block where the first Ada parameter begins, i.e the first extra parameter for the function result, which exists only in the SYSTEAM Ada System internal calling conventions, is not present in the called function.

- The last aspect which is not handled by the SYSTEAM Ada System is the support of complex types, such as string descriptors, etc. These types can be built in the Ada program by appropriate record declarations, with representation clauses if necessary.

**Summary:** If you use only 4 byte parameters and specify PRAGMA interface(VMS,...) and add an additional first integer parameter which holds the number of parameters, then the procedure call generated by the SYSTEAM Ada System conforms to the VAX procedure calling standard.

**Note:** For parameters which are passed by reference by declaring the formal parameter of type SYSTEM.ADDRESS and passing the parameter by <object>'ADDRESS, the SYSTEAM Ada System typically needs PRAGMA resident to be specified for the actual parameter. This prevents the Optimizer of the SYSTEAM Ada System from holding the value of the parameter in a register or even eliminating it completely, because there is no further access to the parameter - from the Optimizers point of view. (cf. §15.1.2)

The SYSTEAM Ada System does not check the observance of the VAX procedure calling standard. If it is violated the call of the non Ada routine will be erroneous.

The following example shows the intended usage of PRAGMA interface (VMS,...) to call a VMS system routine. First some types with representation specifications and objects of those types are declared. Then the Ada specification of sys\_qio appears and is related through appropriate pragmas to the system service SYS\$QIO. It is assumed that the function is called in the body of the main program. This example further shows the use of interrupt entries with the SYSTEAM Ada System (cf. §16.5.1.2).

```
WITH system, text_io;
```

```
PROCEDURE vms_routine IS
```

```
    readprompt      : CONSTANT := 55;
    m_noecho        : CONSTANT := 64;
    null_address    : CONSTANT system.address := system.address_zero;
```

```
TYPE iosb_type IS
```

```
    RECORD
        condition_value : short_integer;
        transfer_count   : short_integer;
        dev_spec_info    : integer;
    END RECORD;
```

```
FOR iosb_type USE
```

```
    RECORD
        condition_value AT 0 RANGE 0 .. 15;
        transfer_count  AT 0 RANGE 16 .. 31;
        dev_spec_info   AT 4 RANGE 0 .. 31;
    END RECORD;
```

```

chan      : integer;           -- channel number
res       : integer;           -- return code

io_buffer : string (1 .. 80);
PRAGMA resident (io_buffer);

prompt_buffer : string (1 .. 2) := " * ";
PRAGMA resident (prompt_buffer);

iosb      : iosb_type;
PRAGMA resident (iosb);

FUNCTION sys_qio (n      : integer := 12; -- number of parameters
                  efn    : integer := 0; -- event flag number
                  chan    : integer;      -- channel
                  func    : integer;      -- function code
                  iosb    : system.address := null_address;
                                      -- IO status block
                  astadr  : system.address := null_address;
                                      -- AST routine
                  astprm  : integer := 0;
                  p1      : system.address; -- IO buffer
                  p2      : integer := 0;  -- IO buffer length
                  p3      : integer := 0;  -- timeout
                  p4      : integer := 0;  -- read terminator
                  p5      : system.address := null_address;
                                      -- prompt buffer
                  p6      : integer := 0)  -- prompt buffer length
RETURN integer;

PRAGMA interface (vms, sys_qio);
-- and additionally
PRAGMA external_name ("SYS$QIO", sys_qio);

vector_number : CONSTANT := 0;
ast_parameter  : CONSTANT := 123_456_789;

TASK buffer_handler IS
  PRAGMA priority (15);

  ENTRY buffer_read (param : integer);
  FOR buffer_read USE AT system.interrupt_vector (vector_number);
END buffer_handler;

TASK BODY buffer_handler IS
BEGIN
  ACCEPT buffer_read (param : integer) DO

```

```

        text_io.put_line (io_buffer);
    END buffer_read;

    END buffer_handler;
BEGIN

    -- after an appropriate ASSIGN channel call:

    res :=
        sys_qio (chan => chan,
            func => readprompt + m_noecho,
            iosb => iosb'address,
            astadr => system.ast_service (vector_number),
            astprm => ast_parameter,
            p1 => io_buffer'address,
            p2 => io_buffer'length,
            p3 => 0,
            p4 => 0,
            p5 => prompt_buffer'address,
            p6 => prompt_buffer'length);
END vms_routine;

```

## 15.2 Implementation-Dependent Attributes

The name, type and implementation-dependent aspects of every implementation-dependent attribute is stated in this section.

### 15.2.1 Language-Defined Attributes

The name and type of all the language-defined attributes are as given in the LRM. We note here only the implementation-dependent aspects.

#### ADDRESS

If this attribute is applied to an object for which storage is allocated, it yields the address of the first storage unit that is occupied by the object.

If it is applied to a subprogram or to a task, it yields the address of the entry point of the subprogram or task body.

If it is applied to a task entry for which an address clause is given, it yields the address given in the address clause.

For any other entity this attribute is not supported and will return the value `system.address_zero`.

### IMAGE

The image of a character other than a graphic character (cf. LRM(§3.5.5(11))) is the string obtained by replacing each italic character in the indication of the character literal (given in the LRM(Annex C(13))) by the corresponding upper-case character. For example, `character'image(nul) = "NUL"`.

### MACHINE\_OVERFLOW

Yields true for each real type or subtype.

### MACHINE\_ROUND

Yields true for each real type or subtype.

### STORAGE\_SIZE

The value delivered by this attribute applied to an access type is as follows:

If a length specification (`STORAGE_SIZE`, see §16.2) has been given for that type (static collection), the attribute delivers that specified value.

In case of a dynamic collection, i.e. no length specification by `STORAGE_SIZE` given for the access type, the attribute delivers the number of storage units currently allocated for the collection. Note that dynamic collections are extended if needed. If the collection manager (cf. §13.3.1) is used for a dynamic collection the attribute delivers the number of storage units currently allocated for the collection. Note that in this case the number of storage units currently allocated may be decreased by release operations.

The value delivered by this attribute applied to a task type or task object is as follows:

If a length specification (`STORAGE_SIZE`, see §16.2) has been given for the task type, the attribute delivers that specified value; otherwise, the default value is returned.

### 15.2.2 Implementation-Defined Attributes

There are no implementation-defined attributes.

### 15.3 Specification of the Package SYSTEM

The package system as required in the LRM (§13.7) is reprinted here with all implementation-dependent characteristics and extensions filled in.

PACKAGE system IS

TYPE designated\_by\_address IS LIMITED PRIVATE;

TYPE address IS ACCESS designated\_by\_address;  
FOR address's storage\_size USE 0;

address\_zero : CONSTANT address := NULL;

TYPE name IS (vax\_vms);

system\_name : CONSTANT name := vax\_vms;

storage\_unit : CONSTANT := 8;  
memory\_size : CONSTANT := 2 \*\* 31;  
min\_int : CONSTANT := - 2 \*\* 31;  
max\_int : CONSTANT := 2 \*\* 31 - 1;  
max\_digits : CONSTANT := 33;  
max\_mantissa : CONSTANT := 31;  
fine\_delta : CONSTANT := 2.0 \*\* (-31);  
tick : CONSTANT := 0.01;

SUBTYPE priority IS integer RANGE 0 .. 15;

FUNCTION "+" (left : address; right : integer) RETURN address;

FUNCTION "+" (left : integer; right : address) RETURN address;

FUNCTION "-" (left : address; right : integer) RETURN address;

FUNCTION "-" (left : address; right : address) RETURN integer;

SUBTYPE external\_address IS STRING;

-- External addresses use hexadecimal notation with characters  
-- '0'..'9', 'a'..'f' and 'A'..'F'. For instance:  
-- "7FFFFFFFF"  
-- "80000000"

```

--      "8" represents the same address as "00000008"

FUNCTION convert_address (addr : external_address) RETURN address;

    -- CONSTRAINT_ERROR is raised if the external address ADDR
    -- is the empty string, contains characters other than
    -- '0'..'9', 'a'..'f', 'A'..'F' or if the resulting address
    -- value cannot be represented with 32 bits.

FUNCTION convert_address (addr : address) RETURN external_address;

    -- The resulting external address consists of exactly 8
    -- characters '0'..'9', 'A'..'F'.

TYPE interrupt_number IS RANGE 0 .. 31;

TYPE interrupt_addresses IS ARRAY (interrupt_number) OF address;

ast_service,
interrupt_vector : interrupt_addresses;

non_ada_error      : EXCEPTION;

-- non_ada_error is raised, if some event occurs which does not
-- correspond to any situation covered by Ada, e.g.:
--   illegal instruction encountered
--   error during address translation
--   illegal address

TYPE exception_id IS NEW address;

no_exception_id      : CONSTANT exception_id := address_zero;

-- Coding of the predefined exceptions:

constraint_error_id : CONSTANT exception_id := ... ;
numeric_error_id    : CONSTANT exception_id := ... ;
program_error_id     : CONSTANT exception_id := ... ;
storage_error_id     : CONSTANT exception_id := ... ;
tasking_error_id     : CONSTANT exception_id := ... ;

non_ada_error_id     : CONSTANT exception_id := ... ;

status_error_id      : CONSTANT exception_id := ... ;
mode_error_id        : CONSTANT exception_id := ... ;
name_error_id        : CONSTANT exception_id := ... ;
use_error_id         : CONSTANT exception_id := ... ;

```

```

device_error_id      : CONSTANT exception_id := ... ;
end_error_id         : CONSTANT exception_id := ... ;
data_error_id        : CONSTANT exception_id := ... ;
layout_error_id      : CONSTANT exception_id := ... ;

time_error_id        : CONSTANT exception_id := ... ;

TYPE argument_array IS ARRAY (1 .. 4) OF integer;

no_condition_name    : CONSTANT := 1;

TYPE exception_information IS
  RECORD
    excp_id           : exception_id;

    -- Identification of the exception. The codings of
    -- the predefined exceptions are given above.

    code_addr         : address;

    -- Code address where the exception occurred. Depending
    -- on the kind of the exception it may be address of
    -- the instruction which caused the exception, or it
    -- may be the address of the instruction which would
    -- have been executed if the exception had not occurred.

    condition_name    : integer;

    -- If /= no_condition_name, the exception was caused
    -- by a condition. In this case, the condition name
    -- and other following information made available.

    nr_of_arguments   : integer;    -- in the range 1 .. 4.

    arguments         : argument_array;

    -- Only arguments (1 .. nr_of_arguments) are valid.
    -- It contains a copy of the optional information
    -- supplied by VMS in the argument array when the
    -- condition occurred. If there are more than 4 optional
    -- entries in the argument array, only the first 4
    -- are copied.

    ps1               : integer;

    -- The processor status longword.

```



```
END RECORD;

PROCEDURE get_exception_information
    (excp_info : OUT exception_information);

    -- The subprogram get_exception_information must only be called
    -- from within an exception handler BEFORE ANY OTHER EXCEPTION
    -- IS RAISED. It then returns the information record about the
    -- actually handled exception.
    -- Otherwise, its result is undefined.

TYPE exit_code IS NEW integer;

-- The exit codes WARNING, ERROR and SEVERE_ERROR set the bit 28,
-- which inhibits the display of the error message 0 by the DCL
-- interpreter
warning      : CONSTANT exit_code := 16#100000000#;
success      : CONSTANT exit_code := 16#000000001#;
error        : CONSTANT exit_code := 16#100000002#;
information  : CONSTANT exit_code := 16#000000003#;
severe_error : CONSTANT exit_code := 16#100000004#;
PROCEDURE set_exit_code (val : exit_code);

    -- Specifies the exit code which is returned to the
    -- operating system if the Ada program terminates normally.
    -- The default exit code is 'success'. If the program is
    -- abandoned because of an exception, the exit code is
    -- 'error'.

PRIVATE

    -- private declarations

END system;
```

## 15.4 Restrictions on Representation Clauses

See Chapter 16 of this manual.

## 15.5 Conventions for Implementation-Generated Names

There are implementation generated components but these have no names. (cf. §16.4 of this manual).

## 15.6 Expressions in Address Clauses

See §16.5 of this manual.

## 15.7 Restrictions on Unchecked Conversions

The implementation supports unchecked type conversions for all kinds of source and target types with the restriction that the target type must not be an unconstrained array type. The result value of the unchecked conversion is unpredictable, if

`target_type'SIZE > source_type'SIZE`

## 15.8 Characteristics of the Input-Output Packages

The implementation-dependent characteristics of the input-output packages as defined in the LRM(Chapter 14) are reported in Chapter 17 of this manual.

## 15.9 Requirements for a Main Program

A main program must be a parameterless library procedure. This procedure may be a generic instantiation; the generic procedure need not be a library unit.

## 15.10 Unchecked Storage Deallocation

The generic procedure `unchecked_deallocation` is provided; the effect of calling an instance of this procedure is as described in the LRM (§13.10.1).

The implementation also provides an implementation-defined package `collection_manager`, which has advantages over unchecked deallocation in some applications (cf. §13.3.1).

Unchecked deallocation and operations of the `collection_manager` can be combined as follows:

- `collection_manager.reset` can be applied to a collection on which unchecked deallocation has also been used. The effect is that storage of all objects of the collection is reclaimed.
- After the first `unchecked_deallocation (release)` on a collection, all following calls of `release (unchecked deallocation)` until the next `reset` have no effect, i.e. storage is not reclaimed.
- after a `reset` a collection can be managed by `mark` and `release` (resp. `unchecked_deallocation`) with the normal effect even if it was managed by `unchecked_deallocation` (resp. `mark` and `release`) before the `reset`.

## 15.11 Machine Code Insertions

A package `machine_code` is not provided and machine code insertions are not supported.

## 15.12 Numeric Error

The predefined exception `numeric_error` is never raised implicitly by any predefined operation; instead the predefined exception `constraint_error` is raised.



## 16 Appendix F: Representation Clauses

In this chapter we follow the section numbering of Chapter 13 of the LRM and provide notes for the use of the features described in each section.

### 16.1 Pragmas

#### **PACK**

As stipulated in the LRM (§13.1), this pragma may be given for a record or array type. It causes the Compiler to select a representation for this type such that gaps between the storage areas allocated to consecutive components are minimized. For components whose type is an array or record type the PRAGMA PACK has no effect on the mapping of the component type. For all other component types the Compiler will choose a representation for the component type that needs minimal storage space (packing down to the bit level). Thus the components of a packed data structure will in general not start at storage unit boundaries.

#### **BYTE\_PACK**

This is an implementation-defined pragma which takes the same argument as the predefined language PRAGMA PACK and is allowed at the same positions. For components whose type is an array or record type the PRAGMA BYTE\_PACK has no effect on the mapping of the component type. For all other component types the Compiler will try to choose a more compact representation for the component type. But in contrast to PRAGMA PACK all components of a packed data structure will start at storage unit boundaries and the size of the components will be a multiple of `system.storage_unit`. Thus, the PRAGMA BYTE\_PACK does not effect packing down to the bit level (for this see PRAGMA PACK).

## 16.2 Length Clauses

### SIZE

for all integer, fixed point and enumeration types the value must be  $\leq 32$ ;  
for `short_float` types the value must be  $= 32$  (this is the amount of storage which is associated with these types anyway);  
for `float` and `long_float` types the value must be  $= 64$  (this is the amount of storage which is associated with these types anyway).  
for `long_long_float` types the value must be  $= 128$  (this is the amount of storage which is associated with these types anyway);  
for access types the value must be  $= 32$  (this is the amount of storage which is associated with these types anyway).  
If any of the above restrictions are violated, the Compiler responds with a `RESTRICTION` error message in the Compiler listing.

### STORAGE\_SIZE

Collection size: If no length clause is given, the storage space needed to contain objects designated by values of the access type and by values of other types derived from it is extended dynamically at runtime as needed. If, on the other hand, a length clause is given, the number of storage units stipulated in the length clause is reserved, and no dynamic extension at runtime occurs.

Storage for tasks: The memory space reserved for a task is 10K bytes if no length clause is given (cf. Chapter 14). If the task is to be allotted either more or less space, a length clause must be given for its task type, and then all tasks of this type will be allotted the amount of space stipulated in the length clause (the activation of a small task requires about 1.4K bytes). Whether a length clause is given or not, the space allotted is not extended dynamically at runtime.

### SMALL

there is no implementation-dependent restriction. Any specification for `SMALL` that is allowed by the LRM can be given. In particular those values for `SMALL` are also supported which are not a power of two.

## 16.3 Enumeration Representation Clauses

The integer codes specified for the enumeration type have to lie inside the range of the largest integer type which is supported; this is the type `integer` defined in package `standard`.

## 16.4 Record Representation Clauses

Record representation clauses are supported. The value of the expression given in an alignment clause must be 0, 1, 2 or 4. If this restriction is violated, the Compiler responds with a **RESTRICTION** error message in the Compiler listing. If the value is 0 the objects of the corresponding record type will not be aligned, if it is 1, 2 or 4 the starting address of an object will be a multiple of the specified alignment.

The number of bits specified by the range of a component clause must not be greater than the amount of storage occupied by this component. (Gaps between components can be forced by leaving some bits unused but not by specifying a bigger range than needed.) Violation of this restriction will produce a **RESTRICTION** error message.

There are implementation-dependent components of record types generated in the following cases :

- If the record type includes variant parts and if it has either more than one discriminant or else the only discriminant may hold more than 256 different values, the generated component holds the size of the record object.
- If the record type includes array or record components whose sizes depend on discriminants, the generated components hold the offsets of these record components (relative to the corresponding generated component) in the record object.

But there are no implementation-generated names (cf. LRM(§13.4(8))) denoting these components. So the mapping of these components cannot be influenced by a representation clause.

## 16.5 Address Clauses

Address clauses are supported for objects declared by an object declaration and for single task entries. If an address clause is given for a subprogram, package or a task unit, the Compiler responds with a **RESTRICTION** error message in the Compiler listing.

If an address clause is given for an object, the storage occupied by the object starts at the given address. Address clauses for single entries are described in §16.5.1.

### 16.5.1 Interrupts

On VAX/VMS, all hardware interrupts are handled by VMS. It is not possible to handle the hardware interrupts directly. However, some system services allow a process to be interrupted when a particular event occurs. Since the interrupt occurs asynchronously, the interrupt mechanism is called an asynchronous system trap (AST) (cf. *VAX/VMS, System Services*). The trap transfers control to a user-specified service routine that handles the event.

VMS delivers an AST when requested to do so, for instance by calls of system services like \$DCLAST, \$ENQ, \$GETDVI, \$GETJPI, \$GETSYI, \$QIO, \$SETIMR or by calls of RMS services. As parameters to a call of a system service the address of the AST service routine (usually the parameter *astadr*) and an AST parameter value (usually the parameter *astprm*) can be specified. The specified AST parameter value is passed to the AST service routine as argument when it is called. In this way one AST service routine can be used to handle several ASTs.

For example, the \$SETIMR service can be used to request an AST after 10 seconds or at 12:00:00. After calling the \$SETIMR service the program continues in its normal control flow. If the time event occurs, VMS causes the normal control flow to be interrupted and executes the AST service routine. Upon completion of the AST service routine the execution of the program is continued where it was interrupted.

An address clause for an entry associates the entry with an AST. When an AST occurs, the AST service routine initiates the entry call; the calling task and the called task continue their execution in parallel.

By this mechanism, an interrupt acts as an entry call to that task; such an entry is called an *interrupt entry*. An interrupt causes the ACCEPT statement corresponding to the entry to be executed.

The AST is mapped to an *ordinary* entry call. The entry may also be called by an Ada entry call statement. However, it is assumed that when an interrupt occurs there is no entry call waiting in the entry queue. Otherwise, the program is erroneous and behaves in the following way:

- If an entry call stemming from an interrupt is already queued, this previous entry call is lost.
- The entry call stemming from the interrupt is inserted into the front of the entry queue, so that it is handled before any entry call stemming from an Ada entry call statement.



### 16.5.1.1 Association between Entry and Interrupt

The association between an entry and an interrupt is achieved via an interrupt number (type `system.interrupt_number`) the range of interrupt numbers being 0 .. 31 (this means that 32 single entries can act as interrupt entries). A single entry of a task which has one IN parameter of type `integer` or `system.address` can be associated with an interrupt number by an address clause (the Compiler does not check these conventions). Since an address value must be given in the address clause, the interrupt number has to be converted into type `system.address`. The array `system.interrupt_vector` is provided for this purpose; it is indexed by an interrupt number to get the corresponding address.

The following example associates the entries `timer_1`, `timer_2`, `io_failure` and `io_success` with the interrupt numbers 10, 11, 12 and 13, respectively.

```
...
TASK handler IS

    ENTRY timer_1 (x : IN integer);
    ENTRY timer_2 (x : IN integer);
    ENTRY io_failure (x : IN system.address);
    ENTRY io_success (x : IN system.address);

    FOR timer_1    USE AT system.interrupt_vector (10);
    FOR timer_2    USE AT system.interrupt_vector (11);
    FOR io_failure USE AT system.interrupt_vector (12);
    FOR io_success USE AT system.interrupt_vector (13);
END;
...
```

The task body contains ordinary accept statements for the entries.

### 16.5.1.2 Association between Interrupt and AST Service Routine

When a system service is called and an entry is to be called via interrupt, the address value `system.ast_service (nr)` must be specified as AST service routine, where `nr` indicates the interrupt number.

The effect of the execution of the AST service routine given by `system.ast_service (nr)` depends on whether there is a task currently waiting at an ACCEPT or selective wait statement for an entry which is associated with the interrupt number `nr`.

If there is a task waiting, a rendezvous with that task is performed immediately. If several tasks are waiting for the same interrupt, the program is erroneous (cf. LRM(§13.5(8))) and a rendezvous is performed with any of these tasks.

Otherwise, the information that the interrupt *nr* occurred and the corresponding AST parameter value are stored by the Ada runtime system. If later on a task performs an **ACCEPT** or selective wait statement for the entry associated with the interrupt *nr* the rendezvous is performed.

Therefore an interrupt is never treated as a conditional entry call. If the interrupt *nr* occurs again before the previous one has been handled, the previous one is lost.

A detailed example for an interrupt entry is given in §15.1.4, in the procedure *vms\_routine*.

### 16.5.1.3 Calling an Asynchronous VMS System Service

The following example shows how to call an entry from an AST delivered by a VMS system service, here the *\$SETIMR* service.

```
...
PROCEDURE vms_setimr ( ... );
PRAGMA external_name (vms_setimr, "SYS$SETIMR");
PRAGMA interface (vms, vms_setimr);
...
vms_setimr -- (1)
  (daytime => ... , -- e.g. 12:00:00.00
   astadr  => system.ast_service (10));

vms_setimr -- (2)
  (daytime => ... , -- e.g. in 10 sec from now on
   astadr  => system.ast_service (11),
   reqidt  => 1234567);
...
```

The effect of the call (1) of *vms\_setimr* is that at the specified time an AST is delivered which is mapped to a call of the entry *timer\_1* of the task specified above. No AST-parameter specified; VMS inserts the default 0, which is passed as argument *x* to the accept body of *timer\_1*.

The effect of the call (2) of *vms\_setimr* is similar. Here the AST-parameter is specified. It is passed as argument *x* to the accept body of *timer\_2*.

#### 16.5.1.4 Calling an Asynchronous RMS Service

The RMS routines are a little bit different from the VMS system services. When calling an RMS service the addresses of two completion routines (one for success, one for failure) can be given as parameters. If this is done, the service is performed asynchronously and upon completion one of the two completion routines is called as AST routine. The address of the FAB (File Access Block) or RAB (Record Access Block) which was passed as operand to the RMS service is passed as AST parameter to the completion routine.

*Example:*

```
...  
PROCEDURE rms_open ( ... );  
  PRAGMA external_name (rms_open, "SYS$OPEN");  
  PRAGMA interface (vms, rms_open);  
...  
my_fab : fab_type;  
...  
rms_open (fab => my_fab'address,  
          err => system.ast_service (12),  
          suc => system.ast_service (13));  
...
```

Upon successful completion of the open operation an AST is delivered which results in calling the entry `io_success` with parameter `x = my_fab'ADDRESS`. If an error occurs, `io_failure` is called instead.

## 16.6 Change of Representation

The implementation places no additional restrictions on changes of representation.



## 17 Appendix F: Input-Output

In this chapter we follow the section numbering of Chapter 14 of the LRM and provide notes for the use of the features described in each section.

### 17.1 External Files and File Objects

The only form of file sharing which is allowed is shared reading. If two or more files are associated with the same external file at one time (regardless of whether these files are declared in the same program or task), all of these (internal) files must be opened with the mode `in_file`. An attempt to open one of these files with a mode other than `in_file` will raise the exception `use_error`.

Files associated with terminal devices (which is only legal for text files) are excepted from this restriction. Such files may be opened with an arbitrary mode at the same time and associated with the same terminal device.

The following restrictions apply to the generic actual parameter for `element_type`:

- input/output of access types is not defined.
- input/output of unconstrained array types is only possible with a variable record format.
- for RMS sequential [relative or indexed] files the size of an object to be input or output must not be greater than 32767 [16383] storage units.
- input/output is not possible for an object whose (sub)type has a size which is not a multiple of `system.storage_unit`. Such objects *can* exist for types for which a representation clause or the PRAGMA pack is given. `Use_error` will be raised by any attempt to read or write such an object or to open or create a file for such a (sub)type.

### 17.2 Sequential and Direct Files

Sequential and direct files are represented by RMS sequential, relative or indexed files with fixed-length or variable-length records. Each element of the file is stored in one record.

### 17.2.1 File Management

Since there is a lot to say about this section, we shall introduce subsection numbers which do not exist in the LRM.

#### 17.2.1.1 The NAME and FORM Parameters

The name parameter string must be a VMS file specification string and must not contain wild cards, even if that would specify a unique file. The function name will return a file specification string (including version number) which is the file name of the file opened or created.

The syntax of the form parameter string is defined by:

```
form_parameter ::= [ form_specification { . form_specification } ]
```

```
form_specification ::= keyword [ => value ]
```

```
keyword ::= identifier
```

```
value ::= identifier | string_literal | numeric_literal
```

For identifier, numeric\_literal, string\_literal see the LRM(Appendix E). Only an integer literal is allowed as numeric\_literal (cf. LRM(§2.4)).

In the following, the form specifications which are allowed for all files are described.

```
ALLOCATION => numeric_literal
```

This value specifies the number of blocks which are allocated initially; it is only used in a create operation and ignored in an open operation. The value of ALLOCATION in the form string returned by the function form specifies the initial allocation size for existing files too.

```
EXTENSION => numeric_literal
```

This value specifies the number of blocks by which a file is extended if necessary; the value 0 means that the RMS default value is taken. For existing files, this value only applies to extensions of the file between open and close operations in the Ada program.

For details see the *VAX/VMS, Record Management Services*.

**MAX\_RECORD\_SIZE => numeric\_literal**

This value specifies the maximum record size in bytes. The value 0 indicates that there is no limit; for direct files, this value is only allowed for indexed files, whereas for sequential files there is no such restriction. This form specification is only allowed for files with variable record format. If the value is specified for an existing file it must agree with the value of the external file.

For files with fixed-length records, the maximum record size equals `element_type'SIZE / system.storage_unit`. If a fixed record format is used, all objects written to a file which are shorter than the maximum record size are filled up with zeros (ASCII.NUL).

**RECORD\_FORMAT => VARIABLE | FIXED**

This form specification is used to specify the record format. If the format is specified for an existing file it must agree with the format of the external file.

#### 17.2.1.2 Sequential Files

A sequential file is represented by an RMS sequential file with either fixed-length or variable-length records (this may be specified by the form parameter).

If a fixed record format is used, all objects written to a file which are shorter than the maximum record size are filled up with zeros (ASCII.NUL).

**END\_OF\_FILE**

If the keyword `END_OF_FILE` is specified for an existing file in an open for an output file, then the file is opened at the end of the file; i.e. the existing file is extended and not rewritten. This keyword is only allowed for an output file; it only has an effect in an open operation and is ignored in a create.

The default form string for a sequential file is :

```
"ALLOCATION      => 0,          EXTENSION      => 0, " &
"RECORD_FORMAT => VARIABLE, MAX_RECORD_SIZE => 0  "
```

The default form may be used for all types (except for those excluded in §17.1).

### 17.2.1.3 Direct Files

The implementation dependent type count defined in the package specification of `direct_io` has an upper bound of :

```
COUNT'LAST = 2_147_483_647 (= INTEGER'LAST)
```

Direct files are represented by RMS sequential files with fixed-length records or by relative or indexed files with either fixed-length or variable-length records. For indexed files, the record index is stored as unsigned four bytes binary value in the first four bytes of each record. If not explicitly specified, the maximum record size equals `element_type'SIZE / system.storage_unit`.

```
BUCKET_SIZE => numeric_literal
```

This value specifies the number of blocks (one block is 512 bytes) for one bucket; the value 0 means that the value is evaluated by RMS to the minimal number of blocks which is necessary to contain one record. The value must be in the range from 0 up to 32. This form specification is only allowed for relative or indexed files. If the value is specified for an existing file it must agree with the value of the external file.

```
ORGANIZATION => INDEXED | RELATIVE | SEQUENTIAL
```



This form specification is used to specify the file organization. If the organization is specified for an existing file it must agree with the organization of the external file.

The default form string for a direct file is :

```
"ALLOCATION    => 0,          EXTENSION    => 0,    " &  
"ORGANIZATION => SEQUENTIAL, RECORD_FORMAT => FIXED"
```

Indexed files with variable-length records and a maximum record size of 0 may be used for all types (except for those excluded in §17.1). Relative files with variable-length records may also be used for all types, but in this case a maximum record size must be specified explicitly. Sequential, relative or indexed files with fixed-length records may not be used for unconstrained array types.

### 17.3 Text Input-Output

Text files are represented as sequential files with variable record format. One line is represented as a sequence of one or more records; all records except for the last one have a length of exactly `MAX_RECORD_SIZE` and a continuation marker (`ASCII.LF`) at the last position. A line of length `MAX_RECORD_SIZE` is represented by one record of this length. A line terminator is not represented explicitly in the external file; the end of a record which is shorter than `MAX_RECORD_SIZE` or which has length exactly `MAX_RECORD_SIZE` and does not have a continuation marker as its last character is taken as a line terminator.

The value `MAX_RECORD_SIZE` may be specified by the form string for an output file and it is taken from the external file for an input file; for an input file, the value 0 stands for the default of 512. For all files which are created, the value `MAX_RECORD_SIZE` is used for the file attribute `MRS` (maximum record size).

A page terminator is represented as a record consisting of a single `ASCII.FF`. A record of length zero is assumed to precede a page terminator if the record before the page terminator is another page terminator or a record of length `MAX_RECORD_SIZE` with a continuation marker at the last position; this implies that a page terminator is preceded by a line terminator in all cases.

A file terminator is not represented explicitly in the external file; the end of the file is taken as a file terminator. A page terminator is assumed to precede the end of the file if there is not explicitly one as the last record of the file. For input from a terminal, a file terminator is represented as `ASCII.SUB` (= `CTRL/Z`).

### 17.3.1 File Management

In the following, the form specifications which are only allowed for text files or have a special meaning for text files are described.

#### CHARACTER\_IO

The predefined package `text_io` was designed for sequential text files; moreover, this implementation always uses sequential files with a record structure, even for terminal devices. It therefore offers no language-defined facilities for modifying data previously written to the terminal (e.g. changing characters in a text which is already on the terminal screen) or for outputting characters to the terminal without following them by a line terminator. It also has no language-defined provision for input of single characters from the terminal (as opposed to lines, which must end with a line terminator, so that in order to input one character the user must type in that character and then a line terminator) or for suppressing the echo on the terminal of characters typed in at the keyboard.

For these reasons, in addition to the input/output facilities with record structured external files, another form of input/output is provided for text files: It is possible to transfer single characters from/to a terminal device. This form of input/output is specified by the keyword `CHARACTER_IO` in the form string. If `CHARACTER_IO` is specified, no other form specification is allowed and the file name must denote a terminal device.

For an infile, the external file (associated with a terminal) is considered to contain a single line. An `ASCII.SUB` (= `CTRL/Z`) character represents a line terminator followed by a page terminator followed by a file terminator. Arbitrary characters (including all control characters except for `ASCII.SUB`) may be read; a character read is not echoed to the terminal.

For an outfile, arbitrary characters (including all control characters and escape sequences) may be written on the external file (terminal). A line terminator is represented as `ASCII.CR` followed by `ASCII.LF`, a page terminator is represented as `ASCII.FF` and a file terminator is not represented on the external file.

Only for input files :

PROMPTING => string\_literal

This string is output on the terminal before an input record is read if the input file is associated with a terminal; otherwise this form specification is ignored.

Only for output files :

MAX\_RECORD\_SIZE => numeric\_literal

This value specifies the maximum length of a record in the external file. Each record which is not the last record of a line has exactly this maximum record size, with a continuation marker (ASCII.LF) at the last position. The value must be in the range 2 .. 512. If a file is created, the specified value (or the default of 512) is used for the file attribute MRS (maximum record size) of the external file. If the value is specified for an existing file it must be identical to the value of the external file.

END\_OF\_FILE

If the keyword END\_OF\_FILE is specified for an existing file in an open for an output file, then the file is opened at the end of the file; i.e. the existing file is extended and not rewritten. This keyword only has an effect in an open operation and is ignored in a create.

The default form string for an input text file is :

"ALLOCATION => 0, EXTENSION => 0, PROMPTING => "" ""

The default form string for an output text file is :

"ALLOCATION => 0, EXTENSION => 0, MAX\_RECORD\_SIZE => 512"

### 17.3.2 Default Input and Output Files

The standard input (resp. output) file is associated with the system default logical names SYS\$INPUT (resp. SYS\$OUTPUT) of VMS. If a program reads from the standard input file, the logical name SYS\$INPUT must denote an existing file. If a program writes to the standard output file, a file with the logical name SYS\$OUTPUT is created if no such file exists; otherwise the existing file is extended.

The qualifiers /INPUT and /OUTPUT may be used for the VMS RUN command to associate VMS files with the standard files of text\_io.

The name and form strings for the standard files are :

```
standard_input  :  NAME => "SYS$INPUT:"
                   FORM => "PROMPTING => "*" " "

standard_output :  NAME => "SYS$OUTPUT:"
                   FORM => "MAX_RECORD_SIZE => 512"
```

### 17.3.3 Implementation-Defined Types

The implementation-dependent types count and field defined in the package specification of text\_io have the following upper bounds :

COUNT'LAST = 2\_147\_483\_647 (= INTEGER'LAST)

FIELD'LAST = 512

## 17.4 Exceptions in Input-Output

For each of `name_error`, `use_error`, `device_error` and `data_error` we list the conditions under which that exception can be raised. The conditions under which the other exceptions declared in the package `io_exceptions` can be raised are as described in the LRM (§14.4).

### NAME\_ERROR

- in an open operation, if the specified file does not exist;
- in a create operation, if the name string contains an explicit version number and the specified file already exists;
- if the name parameter in a call of the create or open procedure is not a legal VMS file specification string; for example, if it contains illegal characters, is too long or is syntactically incorrect; and also if it contains wild cards, even if that would specify a unique file.

### USE\_ERROR

- if an attempt is made to increase the total number of open files (including the two standard files) to more than 18;
- whenever an error occurred during an operation of the underlying RMS system. This may happen if an internal error was detected, an operation is not possible for reasons depending on the file or device characteristics, a size restriction is violated, a capacity limit is exceeded or for similar reasons;
- if the function `name` is applied to a temporary file;
- if the characteristics of the external file are not appropriate for the file type; for example, if the record size of a file with fixed-length records does not correspond to the size of the element type of a `direct_io` or `sequential_io` file. In general it is only guaranteed that a file which is created by an Ada program may be reopened by another program if the file types and the form strings are the same;
- if two or more (internal) files are associated with the same external file at one time (regardless of whether these files are declared in the same program or task), and an attempt is made to open one of these files with mode other than `in_file`. However, files associated with terminal devices (which is only legal for text files) are excepted from this restriction. Such files may be opened with an arbitrary mode at the same time and associated with the same terminal device;
- if a given form parameter string does not have the correct syntax or if a condition on an individual form specification described in §17.2-3 is not fulfilled;
- if an attempt is made to open or create a sequential or direct file for an element type whose size is not a multiple of `system.storage_unit`; or if an attempt is made to read or write an object whose (sub)type has a size which is not a multiple of `system.storage_unit` (such situations can arise for types for which a representation clause or the PRAGMA pack is given);

**DEVICE\_ERROR**

is never raised. Instead of this exception the exception `use_error` is raised whenever an error occurred during an operation of the underlying RMS system.

**DATA\_ERROR**

- the conditions under which `data_error` is raised by `text_io` are laid down in the LRM; the following notes apply to the packages `sequential_io` and `direct_io`:
- by the procedure `read` if the size of a variable-length record in the external file to be read exceeds the storage size of the given variable or else the size of a fixed-length record in the external file to be read exceeds the storage size of the given variable which has exactly the size `element_type'SIZE`.
- In general, the exception `data_error` is not raised by the procedure `read` if the element read is not a legal value of the element type.
- by the procedure `read` if an element with the specified position in a direct file does not exist; this is only possible if the file is associated with a relative or an indexed file.

**17.5 Low Level Input-Output**

We give here the specification of the package `low_level_io`:

**PACKAGE** `low_level_io` **IS**

**TYPE** `device_type` **IS** (`null_device`);

**TYPE** `data_type` **IS**  
         **RECORD**  
             `NULL`;  
         **END RECORD**;

**PROCEDURE** `send_control`      (`device` : `device_type`;  
                                     `data`    : `IN OUT data_type`);

**PROCEDURE** `receive_control` (`device` : `device_type`;  
                                     `data`    : `IN OUT data_type`);

**END** `low_level_io`;

Note that the enumeration type `device_type` has only one enumeration value, `null_device`; thus the procedures `send_control` and `receive_control` can be called, but `send_control` will have no effect on any physical device and the value of the actual parameter `data` after a call of `receive_control` will have no physical significance.